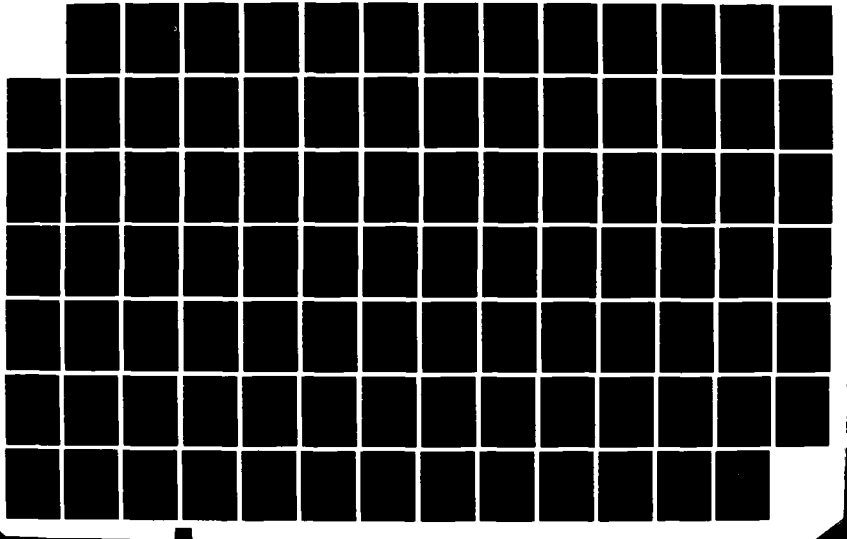
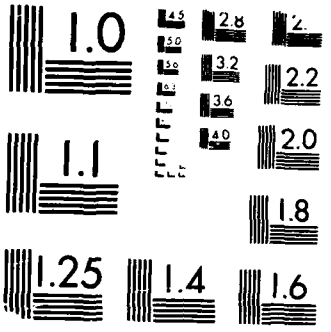
AD-A192 702 INSPECTION METHODS IN PROGRAMMING: CLICHES AND PLANS 1/1
(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB C RICH DEC 87 AI-M-1005
UNCLASSIFIED N00014-85-K-0124 F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1005 December 1987

Inspection Methods in Programming: Clichés and Plans

by

Charles Rich

Abstract

Inspection methods are a kind of engineering problem solving based on the recognition and use of standard forms or *clichés*. Examples are given of program analysis, program synthesis and program validation by inspection. A formalism, called the Plan Calculus, is defined and used to represent programming clichés in a convenient, canonical, and programming-language independent fashion.

Submitted to Artificial Intelligence.

This article describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, in part by the Siemens Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, neither expressed nor implied, of the National Science Foundation, of the IBM Corporation, of the NYNEX Corporation, of the Siemens Corporation, or of the Department of Defense.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AI Memo 1005	2. GOVT ACCESSION NO.: ADA192782
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): Inspection Methods in Programming: Cliches and Plans	5. TYPE OF REPORT & PERIOD COVERED: memorandum
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Charles Rich	8. CONTRACT OR GRANT NUMBER(s): N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Inteligence Laboratory, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: December, 1987
	13. NUMBER OF PAGES: 92
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, VA 22217	15. SECURITY CLASS. (of this report):
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programmer's Apprentice
Automatic Programming
Knowledge Representation
Cliches
Plans
Engineering Problem Solving

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Inspection methods are a kind of engineering problem solving based on the recognition and use of standard forms or cliches. Examples are given of program analysis, program syntheis and program validation by inspection. A formalism, called the Plan Calculus, is defined and used to represent programming cliches in a convenient, canonical, and programming-language independent fashion.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Contents

Acknowledgements

I would like to thank the present and past members of the Programmer's Apprentice group, who have aided the development and refinement of these ideas over a period of years. I especially thank my friend and colleague, Dick Waters, for his careful reading and moral support.

1 Introduction

In textbooks, or in explanations given by experienced engineers and mathematicians, we often encounter the phrase "*by inspection the solution is ....*" This paper begins to develop an account of the role of inspection methods in engineering problem solving generally, and in programming specifically. An important motivation underlying this work is the belief that, in order to further automate the programming process, we must have better computational models of the problem solving methods used by programmers.

The outline of the paper is as follows. In Section 1, engineering problem solving is introduced as a domain of study and is compared with other problem solving domains. Within the engineering context, two very different kinds of problem solving method are contrasted: inspection methods and uniform general methods.

In Section 2, the concept of inspection methods in programming is developed in detail via an extended scenario of analysis by inspection. This section also includes short examples of synthesis by inspection and validation by inspection, which illustrate the shared knowledge (clichés) underlying inspection methods.

Section 3 defines a formalism, called the Plan Calculus, which is used to codify the knowledge underlying inspection methods in programming in a convenient, canonical, and programming-language independent fashion.

Section 4 concludes the paper with a discussion of the relationship of the Plan Calculus to programming languages and other formalisms, current limitations of the Plan Calculus, and further work.

A companion paper [39] describes an initial library of common program forms, which has been compiled using the Plan Calculus, and its use in automated systems for analysis and synthesis of programs.

1.1 Engineering Problem Solving

Programming is viewed here as a kind of engineering activity. This is the appropriate view for understanding the programming involved in the development large software systems.[1] In this context, a first question to ask is: What properties do different kinds of engineering have in common?

[1] The other major school of thought is to view programming as a kind of mathematical activity, which is more appropriate for understanding the development of algorithms.

The first common property of engineering domains is the existence of a set of standardized, well-understood, primitive building blocks. For example, in electrical engineering all circuits are made up at the lowest level of resistances, capacitances, inductances, and so on. Similarly, in mechanical engineering, all devices eventually come down to the primitive mechanisms of lever, gear, rod, pulley, and so on. In software engineering, all programs can be constructed out of assignment, conditional, and recursion. This feature of engineering domains distinguishes them from many other problem solving domains studied in AI (for example, medical diagnosis) in which there is no well-established primitive level of description.

A second common property of engineering domains is that the central problem can be posed abstractly as follows: Given the vocabulary of primitives and the rules for their legitimate combination, devise a composite (usually hierarchical) structure which has some desired behavior. This characterization of engineering problems distinguishes them from other AI problems (for example, playing chess) in which the relationship between structure and function is not the central concern.

In addition to the central synthesis problem, engineers also need to be able to analyze a device (i.e., to infer properties of its behavior from its structure), and incrementally modify (debug) the structure of a device in order to achieve a desired modification in behavior.

In summary, engineering problem solving is concerned with *the analysis, synthesis and debugging of hierarchical objects constructed for an explicit purpose.*

1.2 Uniform General Methods

Two quite different approaches have evolved for solving engineering problems. One approach, which I call *uniform general methods*, takes advantage of the fact that the primitive elements of the domain have well-understood behaviors. For example, in electrical engineering, one way to determine the frequency response of a linear circuit is to solve a set of equations derived from the topology of the circuit, viewed primitively as a network of resistances, capacitances and inductances.

Similarly in mechanical engineering, one way to analyze the stresses and strains in a mechanical structure is by the so-called "finite element method." This method also comes down to solving (usually by computer) a set of

equations derived by viewing the mechanical structure as a grid of primitive geometric elements that interact in simple ways.

Programming also has its uniform general methods. For example, the Floyd-Hoare approach [18, 24] to program verification starts with the semantics of the programming language primitives and combines them according to the structure of the program to derive a single large theorem to be proved (again, usually by computer).

Uniform general methods, such as these examples, have several attractive properties. First, they are based on firm mathematical foundations. As a result, their domain of applicability is well-defined—you know when they will work and when they will not. Second, the solution process is algorithmic, and thus amenable to conventional computerization.

Despite these attractive features, the surpising fact is that experienced engineers typically use uniform general methods only as a last resort. The reason for this is that these methods typically return only an *answer*. They yield little insight into what the engineer is ultimately concerned with, namely the detailed relationship between structure and function in the device under analysis. The engineer needs to understand this relationship in order to modify the structure of the device—for example, to bring it closer to achieving its desired function.

Unfortunately, in real engineering applications (including programming), a detailed description of how the behavior of a composite device follows from the interaction of the behaviors of its primitive components is extremely complex. In response to this complexity, engineering communities have evolved intermediate vocabularies, giving names to those few out of all possible combinations of primitives that have been useful in practice. The next section discusses the kind of problem solving which takes place in an engineering environment that is enriched with this kind of knowledge.

1.3 Inspection Methods

Suppose you present an electrical engineer with a circuit and ask him to answer a question about its behavior, such as: What is the gain (ratio between the strength of the output signal and the strength of the input signal)? One way of answering this question is to employ a uniform general method, namely to methodically translate the structure of the circuit into a corresponding set of equations, which can then be solved to obtain the answer.

This is not, however, the kind of analysis method you are most likely to elicit from an experienced engineer. If the given circuit is designed in accordance with routine engineering practice, an experienced engineer will first *recognize* the circuit. For example, he may say "this is a two-stage audio amplifier." Given this recognition, the task of answering the posed question is greatly simplified. For example, in the case of a two-stage audio amplifer, the engineer knows immediately that the gain may be computed from the product of the ratios of a certain pairs of resistors at key points in the circuit. In electrical engineering, answering questions about a circuit by first recognizing its form is called *analysis by inspection.* Only if you intentionally concoct an obscure circuit, can you force an experienced engineer to resort to setting up equations.

Similarly in programming, suppose you present an experienced programmer with a large data processing system, and enquire as to its maximum running time for given size inputs. Rather than resorting to the first principles of complexity analysis, the experienced programmer will first recognize which of the standard algorithms for searching, sorting, etc. are being employed and then use their known properties to compute the desired property of the net behavior.

There is also *synthesis by inspection.* For example, faced with the task of implementing a common electrical function, such as a high-gain, low-impedance amplifier, the hallmark of an experienced electrical engineer is his ability to retrieve from his mental (or actual) "cook book" an appropriate first-cut design (which he may subsequently modify and refine).[2] Similarly, faced with the task of implementing a common programming behavior, such as associative retrieval, the hallmark of an experienced programmer is his ability to call to mind a repertoire of appropriate standard techniques, such as hashing, discrimination nets, or property lists.

I call these engineering problem solving methods, based on the recognition and use of standard forms, *inspection methods*; I call the standard forms *clichés.* Examples of clichés in the domain of circuits include *voltage divider*, *emitter-coupled pair*, and *Schmidt trigger.* Examples of clichés in the domain of programs include *bubble sort*, *doubly-linked list*, and *linear search.* Clichés form the shared technical vocabulary of a discipline. Although the word cliché has a negative connotation when used in the context of literary

[2]See [46] for a discussion of the role of inspection in relation to abstraction and debugging.

criticism, in engineering, the repeated use of the same "forms of expression" is desirable. Reuse improves productivity in the design process, as well as the understandability (and thus maintainability) of the resulting devices.

A crucial part of any computational account of problem solving in an engineering domain is therefore a representation for the clichés in that domain. In order to motivate the representation for programming clichés introduced in Section 3, Section 2 illustrates the properties and use of programming clichés via several examples.

Notions similar to the cliché idea appear in software engineering in the work of Arango and Freeman [3] (domain models), Harandi and Young [22] (design templates), and Lavi [29] (generic models); and in artificial intelligence in the work of Minsky [35, 36] (frames, concept germs), Schank [50] (scripts), and Chapman [11] (cognitive clichés).

2 Inspection Methods in Programming

The two goals of this section are to deepen the reader's understanding of what is meant by clichés in programming and to motivate the representation for programming clichés defined in Section 3. To achieve these goals, this section presents an informal but detailed scenario of program analysis by inspection.

Solving an analysis problem in the context of programming amounts to deriving some non-obvious properties of a program. To illustrate the role of clichés in this process, let us put ourselves into the following not-so-imaginary situation.

Suppose you are part of the maintenance team for a large software system. You have been assigned a system enhancement task which requires the use of a hash table. In the utilities portion of the system sources, you find the code shown in Figure 1. Unfortunately, as you begin to use this implementation of hash tables in your application, you realize that the documentation doesn't answer an important question: How does this implementation handle duplicate keys? More specifically: If you call `TABLE-INSERT` with an entry whose key might already be in the table, do you first have to call `TABLE-DELETE` to delete the old entry? (Perhaps, in the original application, duplicate keys never occurred, so the implementor didn't think to document what the behavior was under these conditions.)

As a straw man, you might consider solving this analysis problem by formulating it as a theorem—something along the lines of proving that for any table t and entry e,

$$table\text{-}delete\,(table\text{-}insert(t, e), key(e)) = t.$$

If a theorem like this is true, then you can feel free to add and delete entries without worrying about duplicates. If it is not true, however, you need to understand how the proof fails so that you know what aspects of the behavior of `TABLE-INSERT` and `TABLE-DELETE` you can rely upon.

More likely, if you are an experienced programmer, you will take the approach of first studying the code to discover what clichés were used—what is sometimes called "reverse engineering"—and then answering the question of interest based on your understanding of the design. In this example, you know from experience that there are basically two ways to handle duplicate entries in any aggregate structure: either you check for duplicates at insertion time or you search for duplicates at deletion time. The question then boils

```
(DEFUN TABLE-LOOKUP (TABLE KEY)
  (LET ((BUCKET (AREF TABLE (HASH KEY TABLE))))
    (LOOP
      (IF (NULL BUCKET) (RETURN NIL))
      (LET ((ENTRY (CAR BUCKET)))
        (IF (EQUAL (KEY ENTRY) KEY) (RETURN ENTRY)))
      (SETQ BUCKET (CDR BUCKET)))))

(DEFUN TABLE-INSERT (TABLE ENTRY)
  (PUSH ENTRY (AREF TABLE (HASH (KEY ENTRY) TABLE)))
  TABLE)

(DEFUN TABLE-DELETE (TABLE KEY)
  (LET* ((INDEX (HASH KEY TABLE))
         (BUCKET (AREF TABLE INDEX)))
    (IF (EQUAL (KEY (CAR BUCKET)) KEY)
        (SETF (AREF TABLE INDEX) (CDR BUCKET))
        (BUCKET-DELETE BUCKET KEY)))
  TABLE)

(DEFUN BUCKET-DELETE (BUCKET KEY)
  (LET ((PREVIOUS BUCKET))
    (LOOP
      (SETQ BUCKET (CDR PREVIOUS))
      (IF (NULL BUCKET) (RETURN NIL))
      (WHEN (EQUAL (KEY (CAR BUCKET)) KEY)
        (RPLACD PREVIOUS (CDDR PREVIOUS))
        (RETURN NIL))
      (SETQ PREVIOUS BUCKET))))
```

Figure 1. The Common Lisp functions above implement a hash table. Note that the HASH function is not defined here; assume it is just a numerical formula which, although it may also be a cliché, is not the topic of this example. The KEY function simply extracts some field from an entry. There should also be a function for making a new table.

```
(DEFUN TABLE-LOOKUP (TABLE KEY)
  (LET ((BUCKET (AREF TABLE (HASH KEY TABLE))))
    (LOOP
      (IF (NULL BUCKET) (RETURN NIL))
      (LET ((ENTRY (CAR BUCKET)))
        (IF (EQUAL (KEY ENTRY) KEY) (RETURN ENTRY)))
      (SETQ BUCKET (CDR BUCKET)))))
```

linear search

Figure 2. Recognition of linear search cliché.

down to recognizing which (if either) of these two decisions was made in the code. Note also that by taking the approach of understanding the code completely first, you will be in a good position to modify the program to fit your current application, if necessary.

Let us now proceed step by step through an introspective account of recognizing the clichés in the code in Figure 1. As well as introducing further examples of clichés in programming, this scenario also illustrates some important structural aspects of programming clichés which must be addressed in the formal representation.

2.1 Program Analysis by Inspection

We begin with the first function in Figure 1, `TABLE-LOOKUP`. This function is essentially a loop. A key feature of a loop is the number and form of its exit conditions. The loop in `TABLE-LOOKUP` has two exits as indicated in Figure 2. More specifically, this is an instance of the *linear search* cliché:

> A *linear search* is a loop in which a given predicate (the same one each time) is applied to a succession of values (in this case, the values of the variable `ENTRY`) until either: a value is found which satisfies the predicate, in which case the search is terminated and the value satisfying the predicate is made available outside the loop (in this case via `(RETURN ENTRY)`); or there are

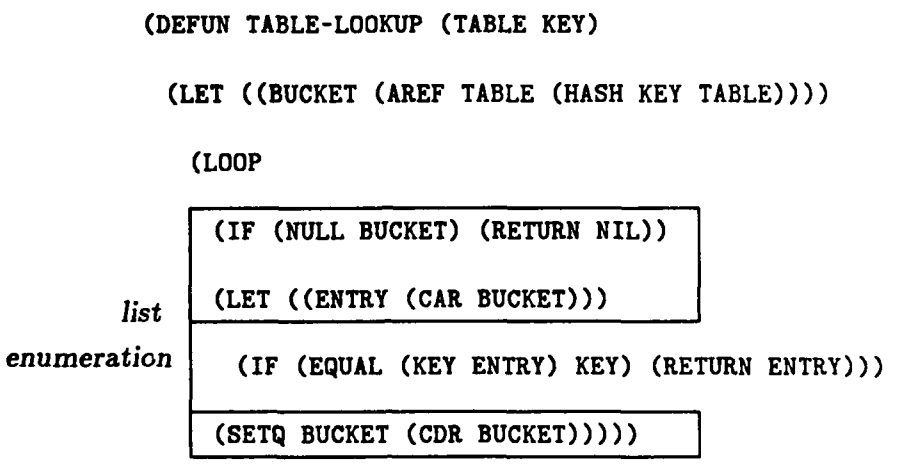


Figure 3. Recognition of list enumeration cliché.

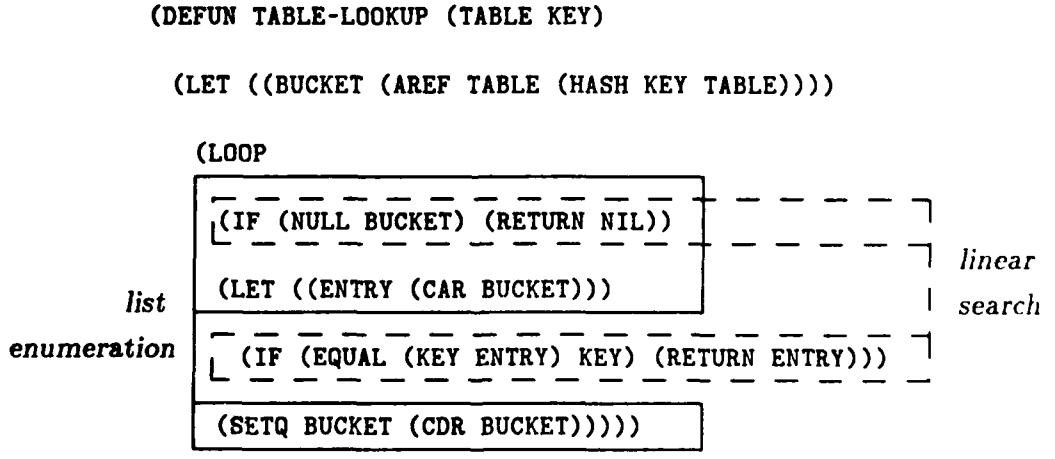


Figure 4. Overlapping occurrences of clichés.

no more values, in which case the search is terminated with a failure indication (in this case, by returning `NIL`).

Figure 3 indicates that `TABLE-LOOKUP` also contains an occurrence of one of the most familiar Lisp programming clichés, namely the `CAR`, `CDR`, `NULL` pattern of *list enumeration*. Note that "pattern" in this context does not mean a particular configuration of the program string or parse tree, but rather a particular set of operations connected by the appropriate data and control flow. In the case of list enumeration, for example, the input to the `NULL` test must be the same as the input to the `CAR` and the `CDR`; and control must exit the loop when the `NULL` test succeeds. The formal representation defined in Section 3 supports this notion of pattern in the definition of programming clichés.

Another important aspect of programming clichés illustrated in `TABLE-LOOKUP` is the fact that occurrences of clichés can overlap. Figure 4 shows the superposition of the linear search and list enumeration clichés recognized above. Notice that the `NULL` exit test fills two roles: it is the failure exit of the linear search and also the empty-list test of the list enumeration. This way of decomposing programs violates the strictly hierarchical approach of most current programming methodologies. We will see several examples, however, in which overlapping decomposition is necessary in order to recognize all the clichés in a program.

The code for `TABLE-INSERT` is only one line long. The only cliché in `TABLE-INSERT` has already migrated into the programming language: The `PUSH` macro in Lisp captures the clichéd use of `CONS` to add an element onto the front of a list, as in

```
(SETQ L (CONS ... L)).
```

(Section 4 discusses the relationship between clichés and programming languages.)

Moving on to `TABLE-DELETE` (Figure 5), we see that the body of this function is a conditional which checks for a common special case that comes up in the implementation of destructive deletion operations, namely deleting the element at the head of the data structure.[3] The form of this cliché, which might be called *special case head deletion*, is as follows:

[3]Failure to check for this special case leads to a characteristic bug. For a further discussion of bug clichés—a topic not pursued in this paper—see [51].

```
(DEFUN BUCKET-DELETE (BUCKET KEY)
  (LET ((PREVIOUS BUCKET))
    (LOOP
      (SETQ BUCKET (CDR PREVIOUS))
      (IF (NULL BUCKET) (RETURN NIL))
      (WHEN (EQUAL (KEY (CAR BUCKET)) KEY)
        (RPLACD PREVIOUS (CDDR PREVIOUS))
        (RETURN NIL))
      (SETQ PREVIOUS BUCKET))))
```

linear search

Figure 5. Recognition of linear search cliché.

```
(DEFUN BUCKET-DELETE (BUCKET KEY)
  (LET ((PREVIOUS BUCKET))
    (LOOP
      (SETQ BUCKET (CDR PREVIOUS))
      (IF (NULL BUCKET) (RETURN NIL))
      (WHEN (EQUAL (KEY (CAR BUCKET)) KEY)
        (RPLACD PREVIOUS (CDDR PREVIOUS))
        (RETURN NIL))
      (SETQ PREVIOUS BUCKET))))
```

trailing pointer list enumeration

Figure 6. Recognition of trailing pointer list enumeration cliché.

```
(DEFUN TABLE-DELETE (TABLE KEY)
  (LET* ((INDEX (HASH KEY TABLE))
         (BUCKET (AREF TABLE INDEX)))
    (IF (EQUAL (KEY (CAR BUCKET)) KEY)
        (SETF (AREF TABLE INDEX) (CDR BUCKET))
        (BUCKET-DELETE BUCKET KEY)))
  TABLE)
```

special case head deletion

Figure 7. Recognition of special case head deletion cliché.

If the head of the data structure (in this case, the CAR of the list BUCKET) satisfies the criterion for deletion (in this case, its KEY is equal to the given key), then update all pointers to the head of the structure to point instead to the tail of the structure (in this case the CDR of the list). Otherwise, if the head of the structure is not to be deleted, use a deletion by side-effect operation which works for "internal" (non-head) elements.

This example illustrates, among other things, that data abstraction needs to be a part of the formalization of programming clichés, since one wants to refer abstractly in the cliché above to the "head" and "tail" of a structure, separate from particular implementations (such as the CAR and CDR of a Lisp list).

Moving on to BUCKET-DELETE (Figure 6), note that this function also contains a linear search. The syntax in this case is very different from the linear search in Figure 2. However, the data and control flow relationships between the two search exits are the same.

BUCKET-DELETE also has instances of the CAR, CDR, and NULL operations with data and control flow between them satisfying the constraints of the list enumeration cliché (see Figure 7). Again, although the syntax of this occurrence of list enumeration is very different from the syntax in TABLE-LOOKUP, we recognize the same cliché. Note that this occurrence of the cliché has an additional bit of structure, which is a common extension of list enumeration,

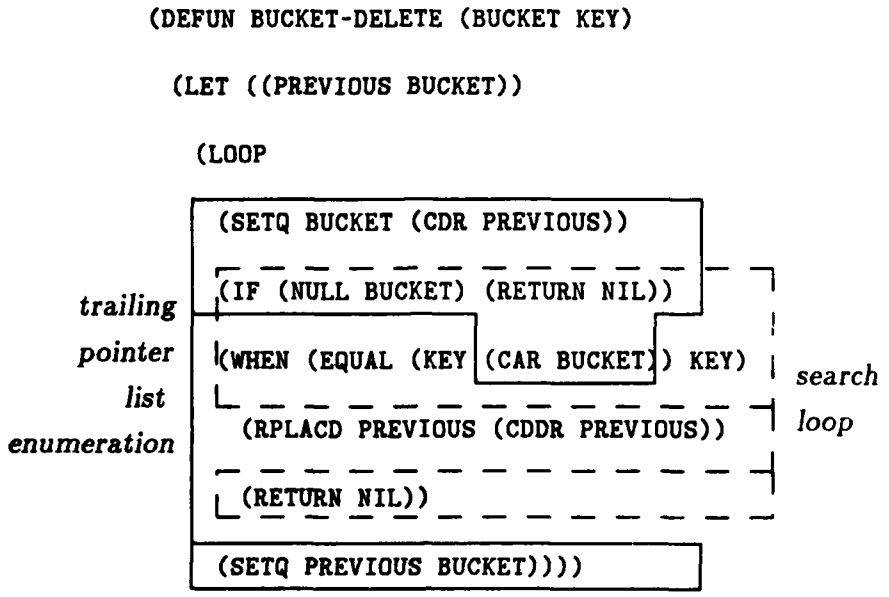


Figure 8. Overlapping linear search and trailing pointer list enumeration clichés.

```
(DEFUN BUCKET-DELETE (BUCKET KEY)
  (LET ((PREVIOUS BUCKET))
    (LOOP
      (SETQ BUCKET (CDR PREVIOUS))
      (IF (NULL BUCKET) (RETURN NIL))
      (WHEN (EQUAL (KEY (CAR BUCKET)) KEY)
        (RPLACD PREVIOUS (CDDR PREVIOUS))      splice out
        (RETURN NIL))
      (SETQ PREVIOUS BUCKET))))
```

Figure 9. Recognition of splice out cliché.

namely a "trailing pointer." The control and data flow in this loop is arranged so that on each iteration there is a pointer (in the variable PREVIOUS) to the cell in the list whose CDR is the current cell being enumerated (in the variable BUCKET). This extension of list enumeration, which might be called *trailing pointer list enumeration* is most commonly used (as is the case here) in connection with destructive deletion operations.

This example illustrates that programming knowledge includes not only clichés, but also relationships between them. *Extension* is one of a number of different relationships between clichés which are supported by the formalism described in the next section.

Finally, note in Figure 8 that the occurrences of the list enumeration and linear search clichés in BUCKET-DELETE overlap in a similar manner to those in TABLE-LOOKUP.

A final cliché that can be recognized in BUCKET-DELETE is *splice out*, as shown in Figure 9. The arbitrary use of side effects like RPLACA and RPLACD can lead to extremely hard-to-understand code. In this case, however, RPLACD is being used in a very specific context: its second argument is the current pointer of a list enumeration and its first argument is the corresponding trailing pointer. This use of RPLACD removes the current element from the enumerated list (by side effect). A cliché like *splice out* is an example of how the recognition of clichés can bypass intractable general-case reasoning.

Using the Results of the Analysis

Now that you have finished analyzing the program, you are in a position to answer the original question quite easily: This implementation does not handle duplicate keys at all, because there is no checking for duplicate keys at insertion time (TABLE-INSERT just does a push) or at deletion time (the linear search cliché used in TABLE-DELETE stops after finding the first value satisfying the criterion). Therefore, you *do* have to call TABLE-DELETE before each call to TABLE-INSERT in which the entry might have a duplicate key.

Furthermore, with this detailed understanding of the relationship between the structure and function of the program, you are also in a good position to modify the program, if desired. For example, suppose you decide to handle duplicate keys at deletion time. There are two changes you need to make to the program.

First, you need to replace the linear search cliché used in BUCKET-DELETE by a related cliché, *exhaustive linear search*, which doesn't stop after find-

ing the first value satisfying the criterion, but rather searches for *all* values satisfying the criterion. The splice out action is then applied to each entry found by the search.

Second, because there could be several duplicate keys at the head of a bucket, the special case head deletion cliché in TABLE-DELETE needs to be replaced by an exhaustive linear search, in which the head deletion action (the SETF) is applied to each case found. (As a code compression, this loop could be combined with the loop in BUCKET-DELETE.)

Viewing the hash table program as the composition of clichés like linear search, splice out, and so on, these changes are modular—a matter of adding or replacing a small number of conceptual parts—even though this may result in many scattered changes at the code level.

2.2 Program Synthesis by Inspection

The notion of recognizing familiar forms applies not only to analysis, but also to the synthesis of programs. For example, consider synthesizing a program to satisfy the following specification: Given a set b and a key k, return a value e, such that

$$(e \in b \ \wedge \ key(e) = k) \ \vee \ (e = nil \ \wedge \ \forall x \in b[key(x) \neq k]).$$

A well-known uniform general method for program synthesis is to treat such a specification as a theorem (literally, $\forall bk \exists e \ldots$). If this theorem can be proved using constructive proof techniques only, then the resulting proof is essentially a program which satisfies the specification.

More likely, if you are an experienced programmer, you will recognize that this specification is not some arbitrary formula in first order logic, but rather an instance of a common specification cliché, which might be called *find if present*: Given an aggregate data structure, find an element satisfying some criterion; or if there is none, return a distinguished value. From experience, this specification suggests the combination of an enumeration with a linear search cliché, as in the following code.

```
(LET ((BUCKET ...))
  (LOOP
    (IF (empty BUCKET) (RETURN NIL))
    (LET ((ENTRY (first BUCKET)))
      (IF (criterion ENTRY) (RETURN ENTRY)))
    (SETQ BUCKET (rest BUCKET))))
```

Enumeration is an abstract cliché comprised of the above pattern of data and control flow between operations on an abstract data type that supports the operations of selecting the first element (*first*), computing an aggregate with all but the first element (*rest*), and testing for empty (*empty*). The linear search cliché, discussed earlier, is comprised of the pattern of data and control flow associated with the *criterion* and *empty* tests above.

The next step in the synthesis is to fill in the criterion role of the linear search with the code for testing the criterion of the specification ($key(e) = k$).

```
(LET ((BUCKET ...))
  (LOOP
    (IF (empty BUCKET) (RETURN NIL))
    (LET ((ENTRY (first BUCKET)))
      (IF (EQUAL (KEY ENTRY) KEY) (RETURN ENTRY)))
    (SETQ BUCKET (rest BUCKET))))
```

To obtain the code for the loop of TABLE-LOOKUP in Figure 1, the final decision to be made is to implement buckets as Lisp lists. This amounts to filling in CAR for *first*, CDR for *rest* and NULL for *empty*.

```
(LET ((BUCKET ...))
  (LOOP
    (IF (NULL BUCKET) (RETURN NIL))
    (LET ((ENTRY (CAR BUCKET)))
      (IF (EQUAL (KEY ENTRY) KEY) (RETURN ENTRY)))
    (SETQ BUCKET (CDR BUCKET))))
```

This example of synthesis by inspection brings out several additional points regarding the formalization of clichés. First, we see that there are standard forms of specifications as well as programs. This suggests a wide-spectrum language, so that the same approach can be applied to both specification and program constructs. Second, we have seen examples of two more kinds of relationships between clichés, namely *implementation* (enumeration and linear search can be used to implement the find if present cliché), and *specialization* (list enumeration is a specialization of enumeration).

2.3 Program Validation by Inspection

Program validation is concerned with making sure that programs do what they are supposed to, or conversely, getting rid of errors.

A uniform general method for program validation is program verification. In this approach, a formal proof is constructed to guarantee that a program satisfies a given formal specification (e.g., a set of preconditions and post-conditions). This is done by combined the specification with the axioms for each language primitive in the program, yielding a single formula/theorem to be proved. This formula can then be passed to a general purpose theorem prover. If the theorem is true, then the program satisfies the specification. Often, however, the theorem is not true, which means that the program has an error. Unfortunately, when this happens, the theorem prover is not in a position to give you much advice as to where in the program the error might be or how to fix it.

Although the ultimate goal of program verification is to confirm that a given program is "correct" with respect to some specification, most of the verification process is actually spent dealing with programs that are not yet correct. What is needed, therefore, is a complementary approach more oriented towards diagnosing errors in terms of the structure of the program, so that the programmer has some hint how to proceed. One such approach is a kind of inspection method that might be called *near-miss* cliché recognition.

Near-miss cliché recognition is based on the idea of near-miss pattern matching, as used by Winston [64] and others. In near-miss recognition, a cliché is recognized when most but not all of its required elements are present. To illustrate, consider the following buggy version of the hash table bucket deletion function.

```
(DEFUN BUCKET-DELETE (BUCKET KEY &AUX PREVIOUS)
  (LOOP
    (IF (NULL BUCKET) (RETURN NIL))
    (WHEN (EQUAL (KEY (CAR BUCKET)) KEY)
      (RPLACD PREVIOUS (CDDR PREVIOUS))
      (RETURN NIL))
    (SETQ PREVIOUS BUCKET)
    (SETQ BUCKET (CDR PREVIOUS))))
```

Under certain input data conditions, this function will cause execution to be interrupted with an error report something like the following.

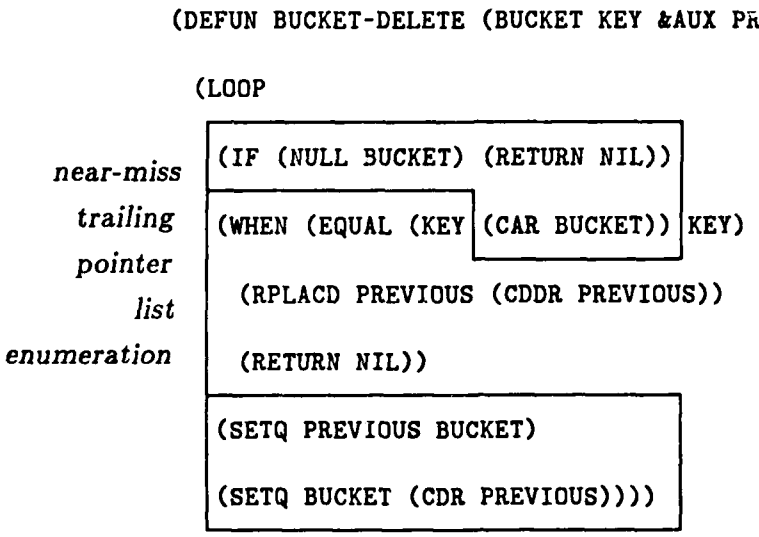


Figure 10. Near-miss recognition of trailing pointer list enumeration cliché.

```
***ERROR***  RPLACD - NIL INVALID ARGUMENT.
```

Applying near-miss cliché recognition to this function (see Figure 10) reveals a near-miss occurrence of the trailing pointer list enumeration cliché. In this definition of BUCKET-DELETE, the appropriate list enumeration operations are present with the appropriate relationships between them, and there is a trailing pointer (in the variable PREVIOUS) whose CDR is the current cell being enumerated (in the variable BUCKET), *except on the first iteration.* Based on this recognition, the following helpful diagnostic message could be produced:

> *It looks like you are trying to implement a trailing pointer list enumeration of* BUCKET, *with* PREVIOUS *as the trailing pointer. Note, however, that on the first iteration of the enumeration, the* CDR *of* PREVIOUS *is not guaranteed to be equal to* BUCKET.

There are, of course, many ways of modifying the program to fix this bug (a correct version is shown in Figure 7). What this example illustrates is that the same knowledge of clichés can be used in many parts of the programming process.

Near-miss cliché recognition is obviously not a complete approach to validation. It does not guarantee that a program does what you want, but only that it does not have a certain class of structural flaws. On the other hand,

this approach does not require you to provide a formal specification, which in many instances is at least as hard to write as the program itself. Furthermore, near-miss cliché recognition, when it works, provides the programmer with a very germane characterization of the error.

An interesting line of research, which is being pursued by Wills [63], is to develop distance metrics which distinguish near-misses that are useful diagnostics, from those that are so far away as to be irrelevant.

This discussion of validation introduces another desideratum for the representation of programming clichés. Since near-miss cliché recognition is not a complete approach, it is desirable to provide a formal semantics for the representation of clichés that will make it possible to apply a combination of inspection methods and more general, theorem-proving methods to validating programs. For example, the synthesis by inspection scenario in the preceding section suggests a verification approach in which a proof structure is built in parallel with the synthesis steps by combining pre-proved lemmas (associated with the clichés), using general theorem-proving as the "glue." This hybrid approach is currently being pursued in a system by Feldman and Rich [44].

3 The Plan Calculus

Formalizing the notion of inspection methods introduced in Sections 1 and 2 has two steps. The first step is to define a representation language for programming clichés. This representation language, called the Plan Calculus, is the topic of this section. The second step is to use the Plan Calculus to codify a library of specific clichés. An initial library of clichés for the routine manipulation of symbolic data is described in a separate paper [39].

3.1 Desired Properties of the Representation

A reader of the scenarios in Section 2 might be left with the impression that a programming cliché could be represented most directly as some fragment of program text, perhaps with holes in it. Although this is an effective expository technique, program text or schemas lack several important properties that are desired in a knowledge representation for clichés, especially for the purpose of building automated programming tools. Three important properties that templates and schemas lack are:

- *Canonical Form*
- *Convenient Manipulation*
- *Language Independence*

A discussion of these properties, and why program text or schemas lack them, serves as a good introduction and motivation for the Plan Calculus.

The first property which program text or schemas lack is canonical form. Consider the linear search cliché as an example. The idea of a linear search could be expressed informally in English as something like the following.

> A *linear search* is a loop in which a given predicate (the same one each time) is applied to a succession of values until either a value is found which satisfies the predicate, in which case that value is made available outside the search; or there are no more values, in which case the search is terminated with a failure indication.

In building a library of clichés, we would like there to be a unique formal structure representing this concept. Unfortunately, in Lisp and most other programming languages, this kind of computation can be written in many different forms, such as:

```
(LOOP
   (IF exhausted (RETURN NIL))
   ...
   (IF (predicate current) (RETURN current))
   ...
   )
```

Or using PROG with only one RETURN, instead of two:

```
(PROG ()
   LP (COND (exhausted NIL)
            (T ...
               (IF (predicate current)
                   (RETURN current))
               ...
               (GO LP))))
```

Or even tail recursively:

```
(DEFUN SEARCH (...)
   (COND (exhausted NIL)
         (T ...
            (COND ((predicate current) current)
                  (T ...
                     (SEARCH ...))))))
```

The problem here is choosing which version to use. Viewed formally as abstract syntax trees in the grammar of the programming language, the different versions above have very different structures. Yet, considering the semantics of the programming language, all three versions specify essentially the same algorithm, i.e., the same set of computations with the same data and control relationships between them.[4]

The Plan Calculus remedies this problem by representing data and control flow structure explicitly. For example, all three of the schemas above (and many other such variations) are canonicalized to the single representation

[4]Some readers may feel that the tail recursive version is fundamentally different. However, recent implementations of Lisp treat loops and tail recursion as alternate stylistic expressions of iteration, i.e., tail recursion is executed without accumulating stack depth.

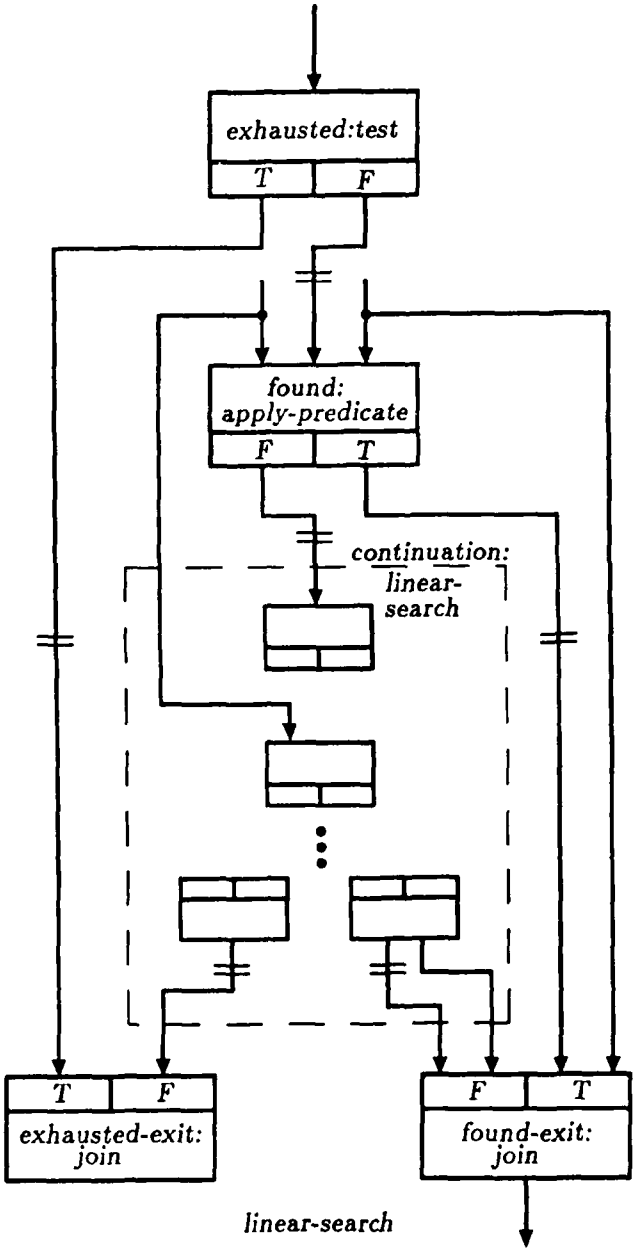


Figure 11. Plan for linear search cliché.

shown in Figure 11.[5] A programming cliché represented in the Plan Calculus, such as Figure 11, is called a *plan*.

The notation used in drawing diagrams of plans is described in detail below. Note for the moment that the formalism takes its inspiration from the kind of diagrams that programmers often scrawl on blackboards and the backs of envelopes when in discussion with other programmers. A plan is essentially a hierarchical graph structure made up of different kinds of boxes and arrows. The inner rectangular boxes denote operations and tests, while the arrows between boxes denote data flow (solid arrows) and control flow (solid arrows with double cross-hatch marks).

A second desired property which program text or schemas lack is convenient manipulation. As anyone who has ever written a complicated macro package can attest, operations on program text, such as concatentation and substitution, are in general a quite tricky business. Typical problems include unintended interactions due to accidental duplication of identifiers and awkward constructions due to mismatch of syntactic forms. Moreover, manipulations which are conceptually simple from an algorithmic point of view often correspond to inconvenient transformations at the program text level. For example, consider combining a cliché of the form

```
(A (B ...) (C ...) ...)
```

with another cliché of the form

```
(F (G ...) (H ...))
```

such that the output of `G` is used as the third input to `A`.

Operating on these clichés in the program text form shown above, this combination is achieved by a complicated sequence of rearrangements resulting in code something like the following:

```
(LET ((X (G ...)))
  (A (B ...) (C ...) X)
  (F X (H ...)))
```

In the Plan Calculus the combination of these two clichés, each expressed as a plan, is a matter of adding only the single data flow arc shown by the

[5] A program which automatically peforms this canonicalization has been implemented by Waters [61].

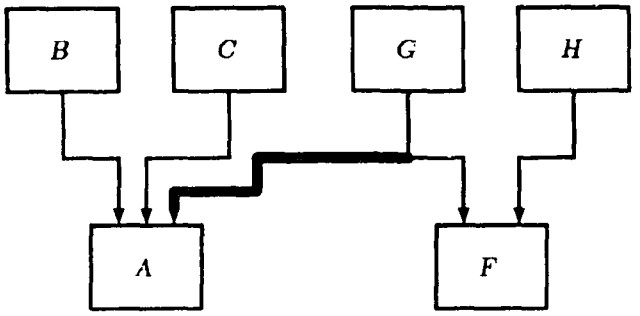


Figure 12. Combining two plans by adding a data flow arc.

bold line in Figure 12. This illustrates how the Plan Calculus is a representation in which the operations that typically occur in the application task (namely, manipulating the algorithmic content of programs) have a more direct correspondence with the operations which are naturally supported by the syntax of the representation (namely, addition, deletion and modification of arcs and nodes in a directed graph).

The use of data flow in the Plan Calculus also reduces the complexity of reasoning about programs by eliminating a lot of spurious side effects. In a conventional programming language semantics, every assignment statement is a side effect. Most assignment statements, however, are not inherently interesting state changes, but rather are part of a pattern of variable assignments and references used to move data from its point of production to its point(s) of use. The Plan Calculus models this use of variables explicitly as data flow arcs.

For example, from a programming-language point of view, the following code involves a side effect (to the variable `X`):

```
(SETQ X (P A))
...
(Q X)
```

The corresponding plan, however, has no side effects in it—the use of the variable `X` corresponds to a data flow arc from `P` to `Q`. (Side effects in the Plan Calculus are discussed further in Section 3.6.)

The third, and most obvious, property that program text or schemas lack is language independence. This is a problem for two reasons. First, from a practical point of view, the compilation of libraries of clichés to support

automated programming tools is likely to be an expensive process, whose cost will need to be amortized over as broad use as possible. A separate library for each programming language makes this amortization more difficult. Second, from a theoretical point of view, common experience tells us that if a programmer knows how to write a cliché like hash table, linear search, or bubble sort in Lisp, he also knows how to do it in other languages in which he is fluent.

The relationship between the Plan Calculus and programming languages is discussed further in Section 4. Modules have been implemented to translate between the Plan Calculus and an assortment of programming languages [14, 59, 61].

An additional desideratum for the representation of clichés is that the formalism be neutral between analysis and synthesis. This turns out to be of practical importance in building interactive programming aids, since in practice these two activities are intermingled. A neutral representation of clichés is also theoretically more attractive than a representation tailored specifically for analysis or synthesis only, since is em a priori a simpler account of the phenomena.

3.2 Plans

The choice of the term *plan* for the knowledge representation used in this work is motivated from two directions. One sense of the term is taken from viewing programming as a kind of engineering activity. Other engineering disciplines have developed specialized schematic languages for representing the structure and function of devices and partial designs. For example, an electrical engineer uses circuit diagrams and block diagrams at various levels of abstraction; a structural engineer uses large-scale and detailed blue prints which show both the architectural framework of a building and also various subsystems such as heating, wiring and plumbing; a mechanical engineer uses overlapping hierarchical descriptions of the interconnections between mechanical parts and assemblies. In this sense, the Plan Calculus is intended to serve as a "blueprint language" for programs. Also, as in other engineering disciplines, the same language is used to describe both specific devices and the clichés out of which these devices are commonly built.

A fundamental characteristic shared by all these types of engineering plans is that at each level there is a set of *parts* with *constraints* between them. Sometimes these parts correspond to discrete physical components,

such as transistors in a circuit diagram. More often, though, the decomposition is in terms of function. For example, a simple amplifier in an electrical block diagram has the functional description $V_2 = kV_1$, where V_1 and V_2 are the input and output signals, and k is the amplification factor. As far as this level of plan is concerned, the amplification may be realized in any number of ways. A primitive component may be used or another plan may be provided which decomposes the amplifier further. By analogy, plans in programming specify the parts of a computation and constraints between them.

Another sense of the term *plan* is taken from the planning subfield of AI. The goal of a planning algorithm is to find a sequence of actions which transforms a given initial state of the world into a desired final state. This problem is analogous to the synthesis of straight-line programs. In early planning work (e.g., [17]), a plan was represented simply as a sequence of actions. Sacerdoti [49], however, showed that it was much more efficient to use a partially-ordered set of actions as the basic representation. In this representation, the planning algorithm needs to consider only those ordering constraints which are actually required by the current set of actions, rather than forcing an arbitrary total ordering. By analogy, a set of operations in the Plan Calculus may be partially ordered by control and data flow, as opposed to operations in program text, which must be totally ordered.

Like plans in the planning literature [48], plans in the Plan Calculus provide representation at different levels of abstraction. Symbolic evaluation of plans in the Plan Calculus [53] is also very similar to techniques used in planning.

Structural and Logical Sublanguages

The Plan Calculus is divided into a *structural* sublanguage and a *logical* sublanguage. The structural sublanguage is the portion of the Plan Calculus shown in plan diagrams. The logical sublanguage comprises the preconditions, postconditions, and other logical statements which annotate the diagrams. Some applications require only the structural part of the Plan Calculus; others also make use of the logical sublanguage.

The following sections undertake the detailed definition of the Plan Calculus in two stages. First a diagrammatic notation for plans is introduced along with an informal description of its semantics in terms of an interpreter for plan diagrams. Following this intuitive introduction, a formal syntax for the structural sublanguage is given.

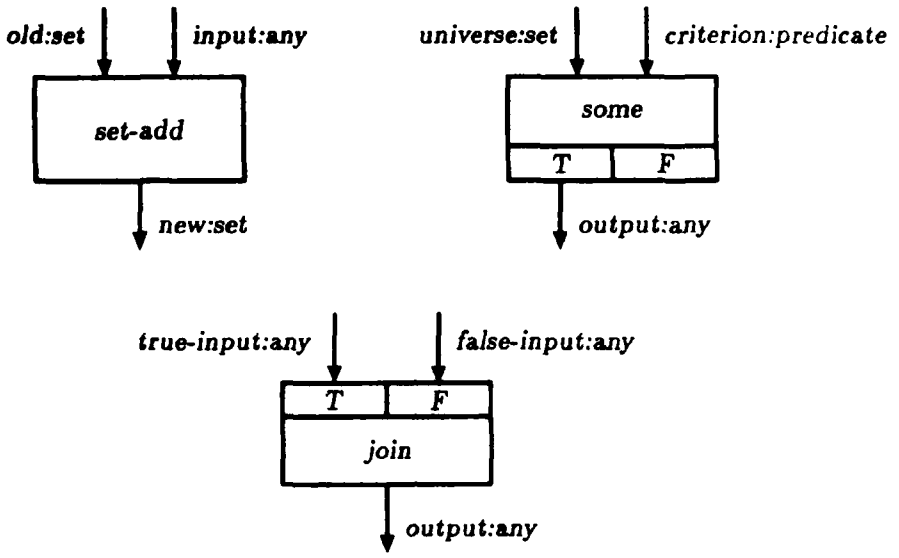


Figure 13. Examples of atomic plan diagram elements: an input/output specification (Set-Add), a test specification (Some), and a join specification (Join).

A formal semantics for the complete language has been developed [40, 41, 42], but is beyond the scope of this paper. The definition of the logical sublanguage is also omitted here, since it is best treated within a larger discussion of the formal semantics.

3.3 Plan Diagrams

A plan diagram is a convenient graphical depiction of the structural portion of the Plan Calculus. Examples of the atomic elements out of which plan diagrams are composed are shown in Figure 13.

Input/Output Specifications

The box labelled Set-Add in Figure 13 is an example of an *input/output specification*. An input/output specification is drawn as a rectangular box with arrows entering at the top (denoting the inputs) and leaving from the bottom (denoting the outputs). Each input or output is labelled with a name and a type, separated by a colon. For example, Set-Add has two inputs: Old (of type Set) and Input (of type Any). The single output of Set-Add is

called New (of type Set). The names of the inputs and outputs within a given input/output specification must be unique. However, the same names may be reused in other specifications.

The logical portion of an input/output specification associates a set of preconditions and postconditions with each plan diagram box. For example, the postconditions of Set-Add state that the New set includes the Input object, all the elements of the Old set, and no others. The logical sublanguage also includes a hierarchy of types, in which Any is defined as the most general data type.

Test Specifications

The box labelled Some in Figure 13 is an example of a *test specification*. A test specification is drawn as a rectangular box, in which the bottom part is divided into two sides labelled "T" and "F". The inputs to a test specification are just like the inputs to an input/output specification. The outputs of a test specification are divided into two groups. Those outputs produced when the test succeeds are indicated leaving from the side of the bottom labelled "T"; those outputs produced when the test fails are indicated leaving from the side of the bottom labelled "F". For example, Some has two inputs: the Universe (of type Set) and the Criterion (of type Predicate). The output of Some, which is defined only when the test succeeds, is called Output (of type Any).

As with input/output specifications, the logical portion of a test specification associates a set of preconditions and postconditions with each plan diagram box. For example, the postconditions of Some state that the Output (when it is produced) is a member of the Universe and that the Criterion is true of it.

Test specifications also include a *test condition*, which is true if and only if the test succeeds. The test condition of Some states that there exists an element of the Universe such that the Criterion is true. (The generalization of test specifications to n mutually exclusive test conditions is straightforward.)

Join Specifications

The box labelled Join in Figure 13 is an example of a *join specification*. A join specification is drawn as a rectangular box with the top part divided into two sides labelled "T" and "F". Inputs to a join specification are indicated

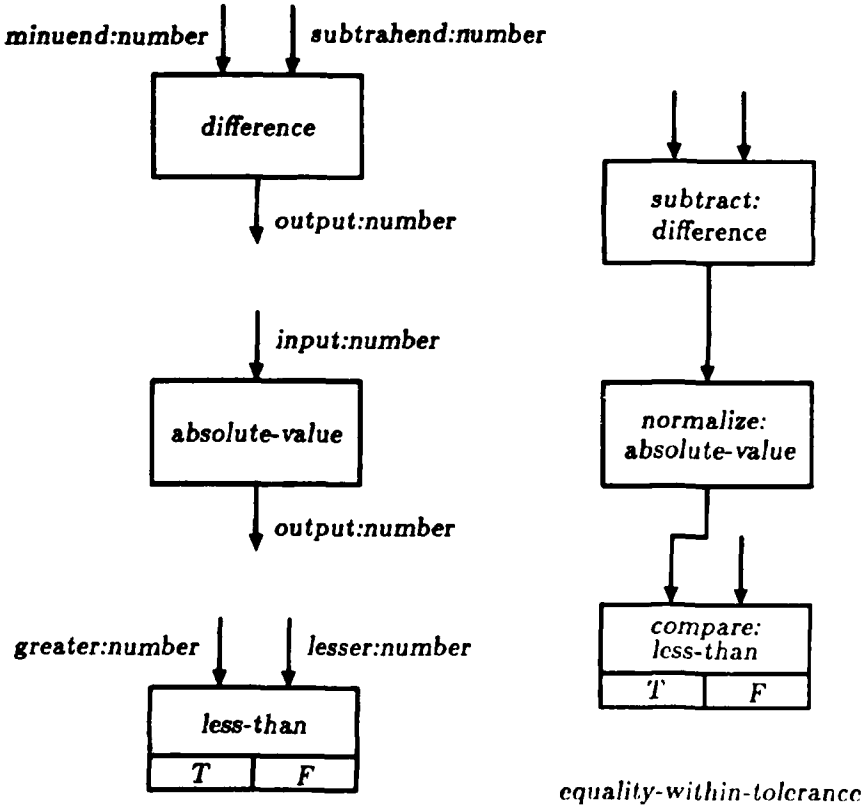


Figure 14. The three input/output and test specifications illustrated at the left of the figure are combined using data flow to construct the plan diagram at the right. Equality-Within-Tolerance checks whether two quantities are equal, within some tolerance.

entering the top of the box; outputs leaving from the bottom. Join specifications are used to end conditional blocks begun by test specifications. The inputs to a join specification are grouped similarly to the outputs of a test specification. The inputs on either the "T" or the "F" side are consumed only when the corresponding branch of the conditional block is executed. The output of a join specification is always the same as whichever input is consumed.

The join specification in Figure 13 has one input on each side and one output. Other join specifications may have several—but the same number of—inputs on each side, and on the output. (Join specifications can be generalized to n mututally exclusive cases analogously to test specifications.)

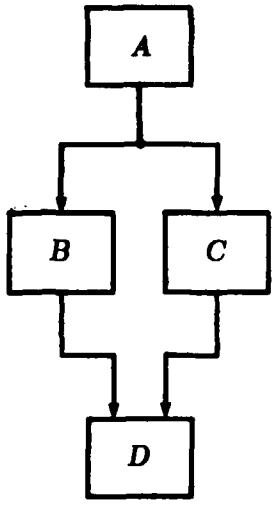
Unlike input/output and test specifications, join specifications do not correspond to any real computation in the final program. Rather, they are a technical artifact used in well-formed plans to specify which data is made available for further computation, depending on which branch of a conditional is executed. For example, the logical conditions associated with Join state that the Output is equal to the True-Input when the "T" case holds, or the False-Input when the "F" case holds.

Data Flow

Input/output specifications, test specifications and join specifications are connected together to form plan diagrams using two kinds of structural constraints.

The first kind of structural constraint is *data flow*. Data flow is shown in plan diagrams by a solid arrow connecting an output of one box with an input of another. Data flow arcs may fan out (i.e., there may be several arcs originating at a given output), but may not fan in (i.e., there may be only one arc terminating at a given input). No directed cycles are allowed (loops are represented using tail recursion, as described below.)

Figure 14 shows a simple plan diagram, Equality-Within-Tolerance, constructed using data flow. Equality-Within-Tolerance checks whether two quantities are equal, within some tolerance. Each box in a plan has a unique name, so that multiple occurrences of boxes of the same type may be referred to unambiguously. These names are called the *roles* of the plan. The roles of the plan Equality-Within-Tolerance are Subtract (of type Difference), Normalize (of type Absolute-Value), and Compare (of type Less-Than). To reduce clutter in plan diagrams, the names and type restrictions of the in-




```
(LET ((X (A...)))
  ...
  (D (B X) (C X)))
```

```
(LET* ((X (A ...))
       (Y (C X)))
  ...
  (D (B X) Y))
```

Figure 15. In the plan diagram at the left, data flow only partially constrains the order of steps in the computation. Both versions of the code at the right are allowed by this plan.

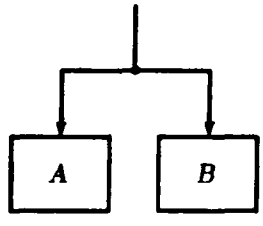


Figure 16. An example of a plan with a data flow constraint in which two inputs are "wired together."

puts and outputs of boxes are usually omitted, since they can be found by reference to the definition of the box type.

Data flow constraints in plan diagrams are an abstraction of the various different mechanisms by which the flow of data can be achieved in different programs and in different languages. These mechanisms include nesting of expressions, use of intermediate variables, and special forms. For example, in the following Lisp code for Equality-Within-Tolerance, all the data flow is achieved by nesting.

```
(< (ABS (- ... ...)) ...)
```

The same data flow could also be coded using an intermediate variable,

```
(LET ((X (- ... ...)))
  ...
  (< (ABS X) ...)))
```

or a combination of nesting, an intermediate variable, and a special form.

```
(LET ((Y (PROG ...
           (RETURN (ABS (- ... ...))))))
  (< Y ...))
```

Data flow constraints also provide another kind of abstraction of program text: Any order of steps is allowed in the final program, as long as it is compatible with (i.e., is a completion of) the partial order specified by the data flow (and the control flow—see the following section). An example of a partially-ordered plan and two final programs is shown in Figure 15.

A slightly different kind of structural feature which can also be thought as a data flow constraint is illustrated in Figure 16. In this plan, the inputs to A and B are "wired together." What this means is that when this plan is combined with other plans, the data flow to A and B must come from the same output. This feature also appears in Figure 11 earlier in this section.

Control Flow

The second kind of structural constraint in plans is *control flow*. Control flow is shown in plan diagrams by a cross-hatched arrow between an *exit point* of one box and an *entry point* of another. Input/output specifications have a single entry point (at the top of the box) and a single exit point (at the bottom of the box). Test specifications have a single entry point (at the top

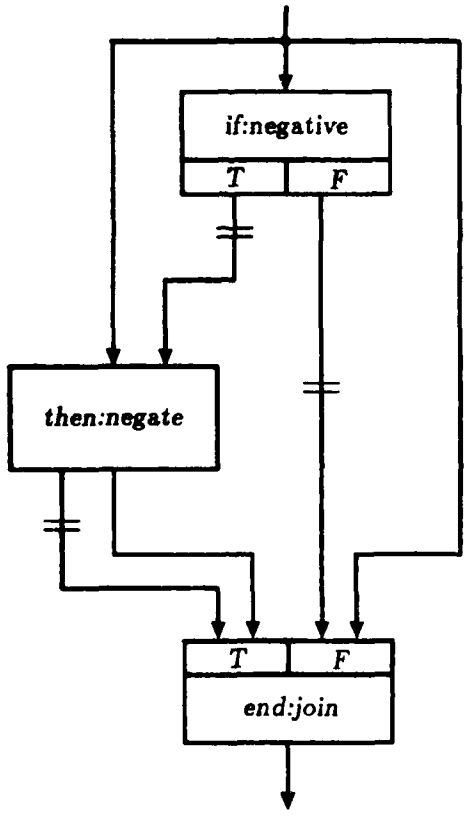
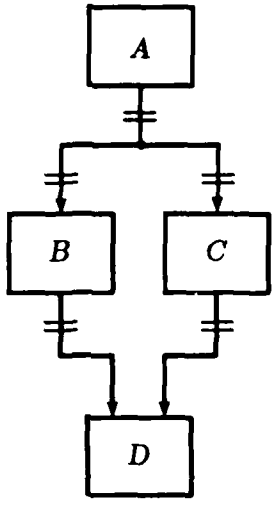


Figure 17. A plan diagram illustrating control flow and data flow. Compute-Absolute-Value computes the absolute value of a number.

of the box) and two exit points (one on each side of the bottom of the box). Join specifications have two entry points (one on each side of the top of the box) and one exit point (at the bottom of the box). Control flow arcs may both fan in and fan out. No directed cycles are allowed.

Figure 17 shows a simple plan diagram, Compute-Absolute-Value, constructed using control flow and data flow. Compute-Absolute-Value computes the absolute value of a number by negating it if necessary.

It is important to note the distinction being made here between Absolute-Value and Compute-Absolute-Value. Absolute-Value is an input/output




```
(PROGN
 (A)
 (B)
 (C)
 (D))

(PROGN
 (A)
 (C)
 (B)
 (D))
```

Figure 18. In the plan diagram at the left, control flow only partially constrains the order of steps in the computation. Both versions of the code at the right are allowed by this plan.

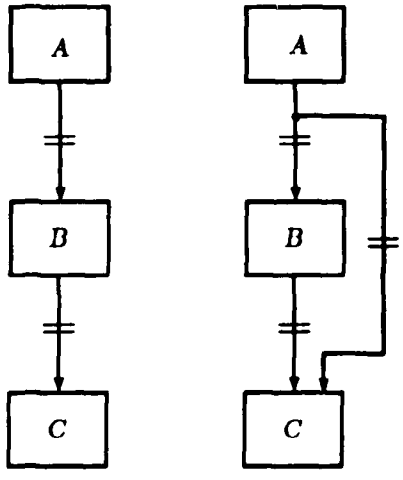


Figure 19. These two plan diagrams have the same meaning, due to the transitivity of control flow.

specification (used, for example, in the plan Equality-Within-Tolerance in Figure 14), which has the postcondition that the output is the absolute value of the input. Compute-Absolute-Value is a plan (combination of steps) which implements this specification. In general, there may be several different plans which implement a given specification. The notion of a plan implementing a specification is captured in overlays, a feature of the Plan Calculus described below.

Control flow constraints in plan diagrams are an abstraction of the various different mechanisms by which the flow of control can be achieved in different programs and in different languages. These mechanisms include nesting of expressions, sequencing primitives, and special forms. For example, in the following Lisp code for Compute-Absolute-Value, the necessary control flow is achieved using the special form **IF**.

```
(IF (MINUSP X)
    (SETQ X (- X)))
```

The same control flow is achieved in a more complicated way in the following code through the interaction of the special forms COND, PROG and RETURN.

```
(PROG ...
   (COND ((MINUSP X) ...)
         (T (RETURN)))
   (SETQ X (- X)))
```

Like data flow, control flow constraints also provide a partial-order abstraction of program text. Conventional programming languages do not distinguish between the necessary orderings between program steps and those that are chosen arbitrarily. In the Plan Calculus, any order of steps is allowed in the final program, as long as it is compatible with (i.e., is a completion of) the partial order specified by the data and control flow. Thus a control flow arc between box A and box B in a plan diagram does not mean that B immediately follows A, but rather than B *eventually follows* A. An example of a plan partially-ordered by control flow and two possible final programs is shown in Figure 18.

Unlike data flow, control flow constraints are transitive. For example, the two plan diagrams in Figure 19 have the same meaning, despite the fact that they have different syntax. This is an undesirable lack of canonicalness in the structural part of the Plan Calculus, which has been handled by building

knowledge of transitivity into the programs which manipulate the diagrams. (See Section 4 for a discussion of other approaches to fixing this problem.)

Finally, note that notion of control flow used here has a different flavor from the notion used in typical flowchart languages. In the Plan Calculus, a control flow arc is a constraint on possible execution orders, whereas in a typical flowchart language, a control flow arc is more like an abstract "jump" instruction.

3.4 A Parallel Execution Model for Plan Diagrams

The meaning of a plan diagram is defined formally as the set of computation sequences it allows (see [40, 41, 42]). A useful intuitive model for plan diagrams, however, is to imagine their direct execution as parallel dataflow programs. This section describes a set of rules for executing plan diagrams. A symbolic interpreter for plan diagrams along these lines was implemented by Shrobe [53].

Basically, plan diagrams are executed by having "tokens" flow between boxes along the data and control flow arcs in a plan. Boxes consume tokens at the top and produce tokens at the bottom. The tokens that flow along data flow arcs are symbolic objects with the appropriate properties. The tokens that flow along control flow arcs are only for controlling conditional execution, and have no other properties. Each box has a buffer for each input, where data tokens wait until they are consumed, and a counter for each entry point, which counts how many control tokens have arrived.

Execution begins by inserting tokens representing the starting data into the input buffers of the data flow sources of the diagram, i.e., the inputs of boxes that have no incoming data flow arcs. Execution then proceeds in parallel according to the activation rules for each kind of box.

An input/output specification is activated when a token has arrived at each of its incoming arcs, i.e., when there is a data token waiting in each of its input buffers, and the entry point has counted a control token for each incoming control flow arc. (If there are no incoming control flow arcs, then this part of the condition is satisfied vacuously.)[6]

[6]Notice that there is a slight asymmetry here between data flow and control flow. Incoming control flow arcs fan-in at a single entry point, whereas each data input is allowed only a single incoming data flow arc. Another way of thinking of this, which resolves the asymmetry, is to consider each incoming control flow arc to have a separate

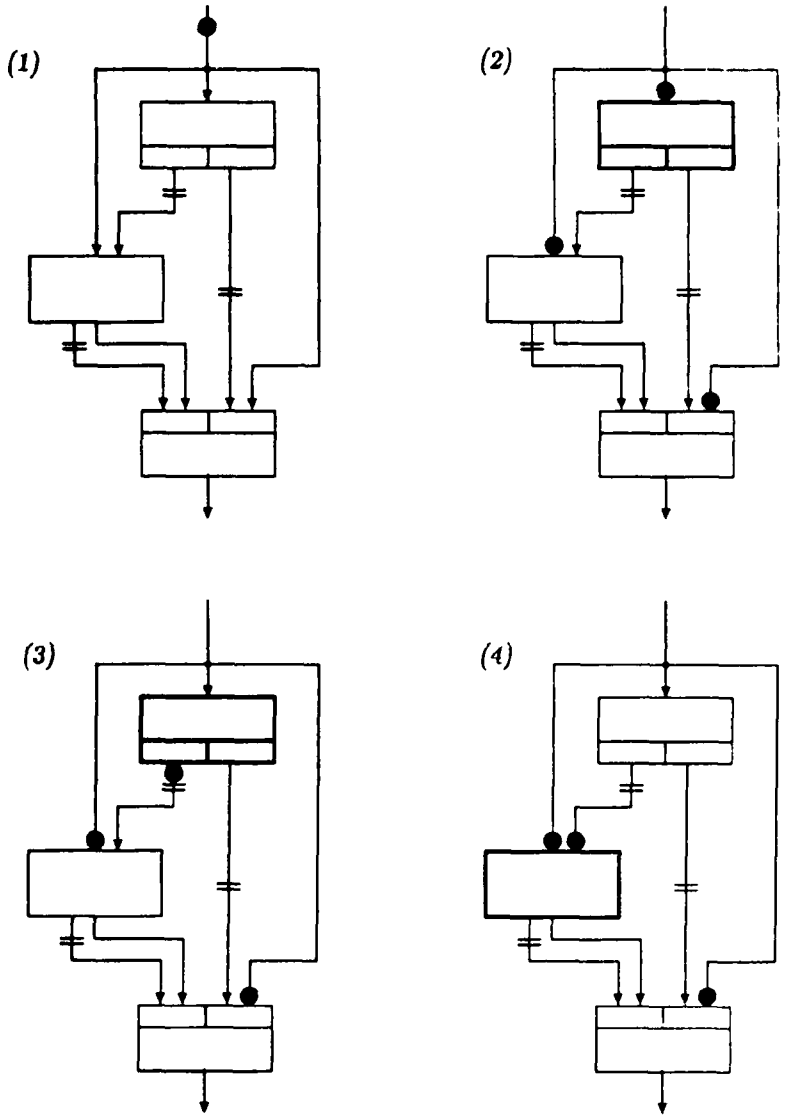


Figure 20. An example of executing the plan diagram for Compute-Absolute-Value (see Figure 17) according to the parallel execution model. The large solid circles represent data and control tokens flowing along arcs. Activated boxes are indicated in bold. The example continues in Figure 21.

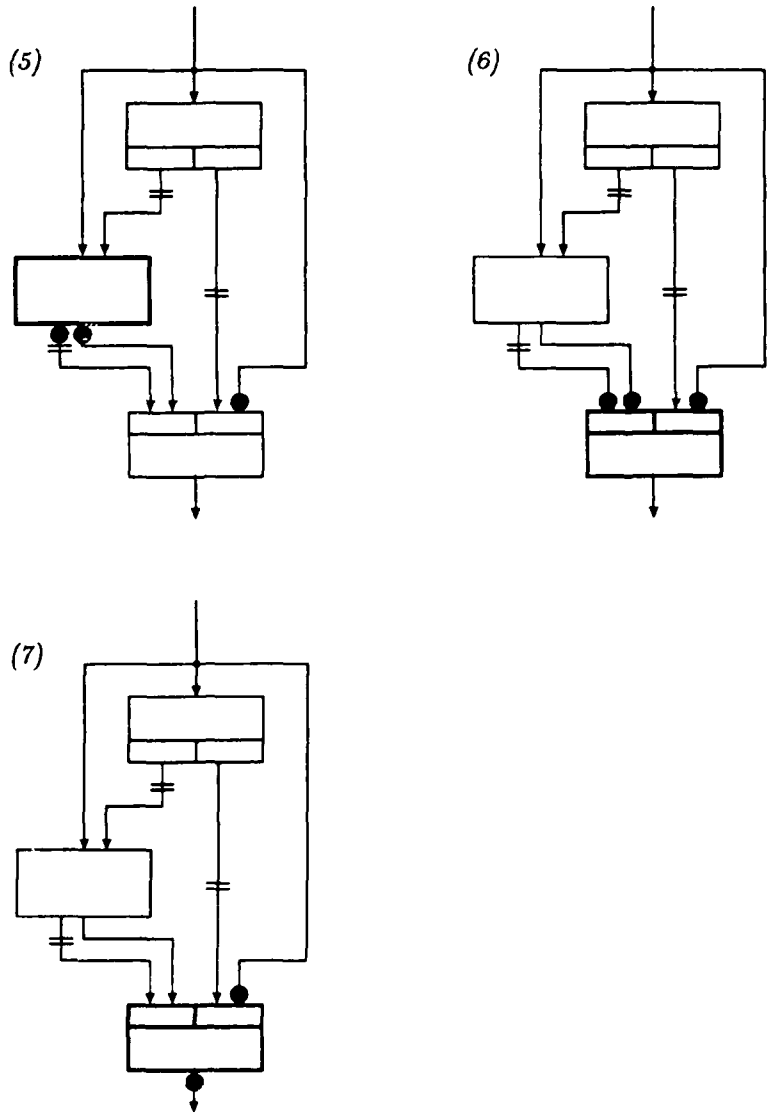


Figure 21. Continuation of Figure 20, showing an example of executing the plan diagram for Compute-Absolute-Value according to the parallel execution model. The large solid circles represent data and control tokens flowing along arcs. Activated boxes are indicated in bold.

When an input/output specification is activated, if the input data satisfies the preconditions of the specification, then output data satisfying the postconditions is produced at each output, and a control token is produced at the exit point. If the input data does not satisfy the preconditions, execution terminates abnormally.

When tokens are produced at an output or exit point, they are propagated along the data flow and control flow arcs to the input buffers and entry point counters of the connected boxes. Where there is fan-out of data flow arcs, the intuitive model is that multiple pointers to the same data are created, as opposed to multiple copies. (This is to allow modelling of side effects - see below.) Where there is fan-out of control flow arcs, it doesn't matter whether you copy or create multiple pointers, since control tokens have no distinct properties.

A test specification is activated the same way as an input/output specification. If the input data does not satisfy the preconditions of the specification, execution is terminated abnormally. If the input data does satisfy the preconditions and the test condition is true, then output data and a control token are produced on the "T" side of the box; otherwise output data and a control token are produced on the "F" side of the box.

A join specification is activated when tokens are present at all of the incoming arcs of one or the other side of the box. When this occurs, a control token is produced at the exit point, and the appropriate data tokens are passed through to the corresponding outputs.

Figures 20 and 21 show an example execution of the Compute-Absolute-Value plan.

A few points are worth noting about this execution model. First, the purpose of the model is to provide intuition into the meaning of the diagrams, not to provide a formal foundation. For example, to formally prove properties of plans (such as whether a given plan terminates normally for all possible inputs) requires manipulating the logical sublanguage of the preconditions, postconditions, and test conditions, which is outside of this execution model.

Second, this execution model is particularly easy to visualize, because there are no cycles in the control flow or data flow. This does not, however, allow for looping computations. The next section introduces hierarchical and recursively defined plans, which are used to model loops and recursive

"control flow input." (The way control flow fan-in is typically drawn in plan diagrams suggests this view.) This view, however, has the undesirable property that the number of control flow inputs is not fixed for a given type of box, but depends on the context of use.

computations generally.

Finally, note that the Plan Calculus is a wide-spectrum language (this point will be discussed further in Section 4). Depending on how specific the input data is, and whether the steps of the plan are totally ordered, executing a plan can range from being equivalent to executing a conventional program, to being the symbolic evaluation of a specification.

3.5 Hierarchical Plans

The type of a role in a plan, in addition to being an atomic element (an input/output, test, or join specification), may also be a plan. This makes it possible to reuse already defined clichés to build larger clichés in a hierarchical fashion. For example, Figure 22 shows the plan Approx-and-Retry-Sqrt, which has the plan Equality-Within-Tolerance (defined earlier in Figure 14) as one of its subplans. Approx-and-Retry-Sqrt is a somewhat contrived plan that computes the square root of a number using an approximation operation and retries the approximation only once if necessary.

Note that the square-root approximation operation (Approx-Sqrt) in Figure 22 has an extra input (Limit) specifying the maximum number of steps to be used in the approximation. If the result of the operation is not within tolerance (Check), the iteration limit is increased (Increase) and the approximation is tried again (Retry). The role Check is itself a plan, Equality-Within-Tolerance, with roles Subtract, Normalize and Compare. Note that the formula used to compute the new, increased limit from the old limit and the absolute value of the error is not specified in this diagram.

Within hierarchical plans, it is convenient to refer to parts at different levels in the structure by composing role names into *paths*. For example, in the plan Approx-and-Retry-Sqrt, the path Approx.Limit refers to the Limit input of the Approx role. Similarly, Check.Compare.Lesser refers to the Lesser input to the Compare role of the Check role.

Notice in Figure 22 that a dashed box is drawn around the parts of a subplan. This boundary is not, however, a barrier to establishing connections between the parts of the subplan and the surrounding plan. Inputs to intermediate steps of a subplan can be provided from the surrounding plan and intermediate results can be "tapped." For example, note the data flow connection between the output of Check.Normalize and the input of the Increase step.

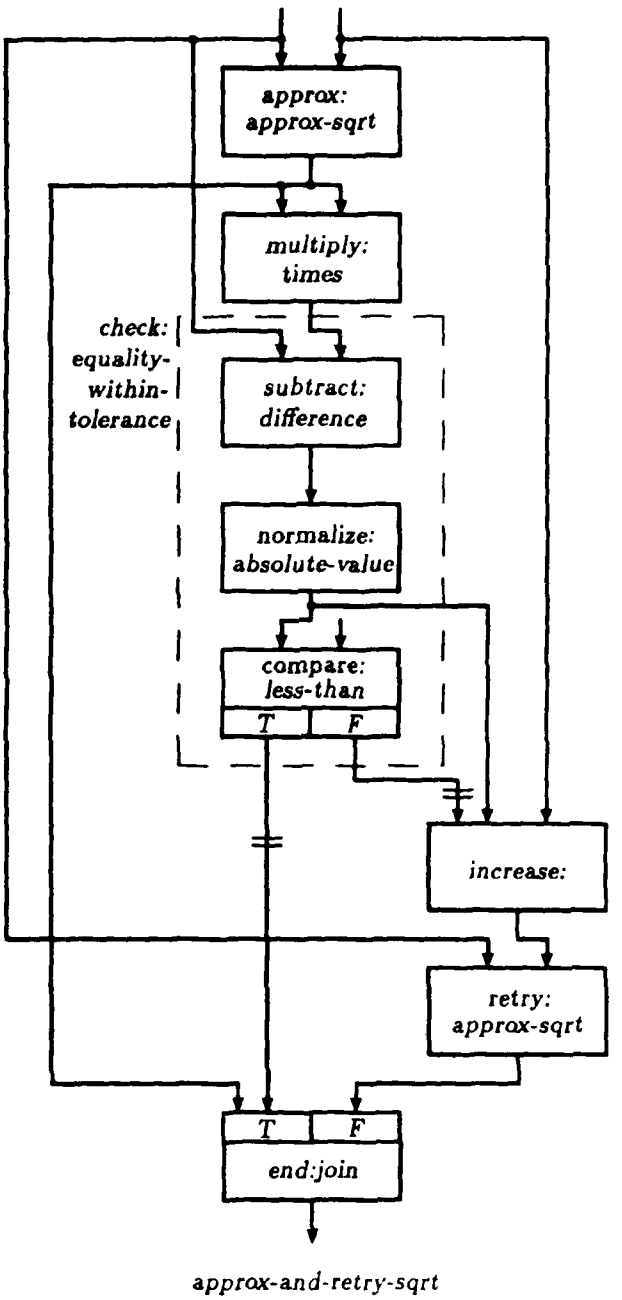


Figure 22. An example of a hierarchical plan.

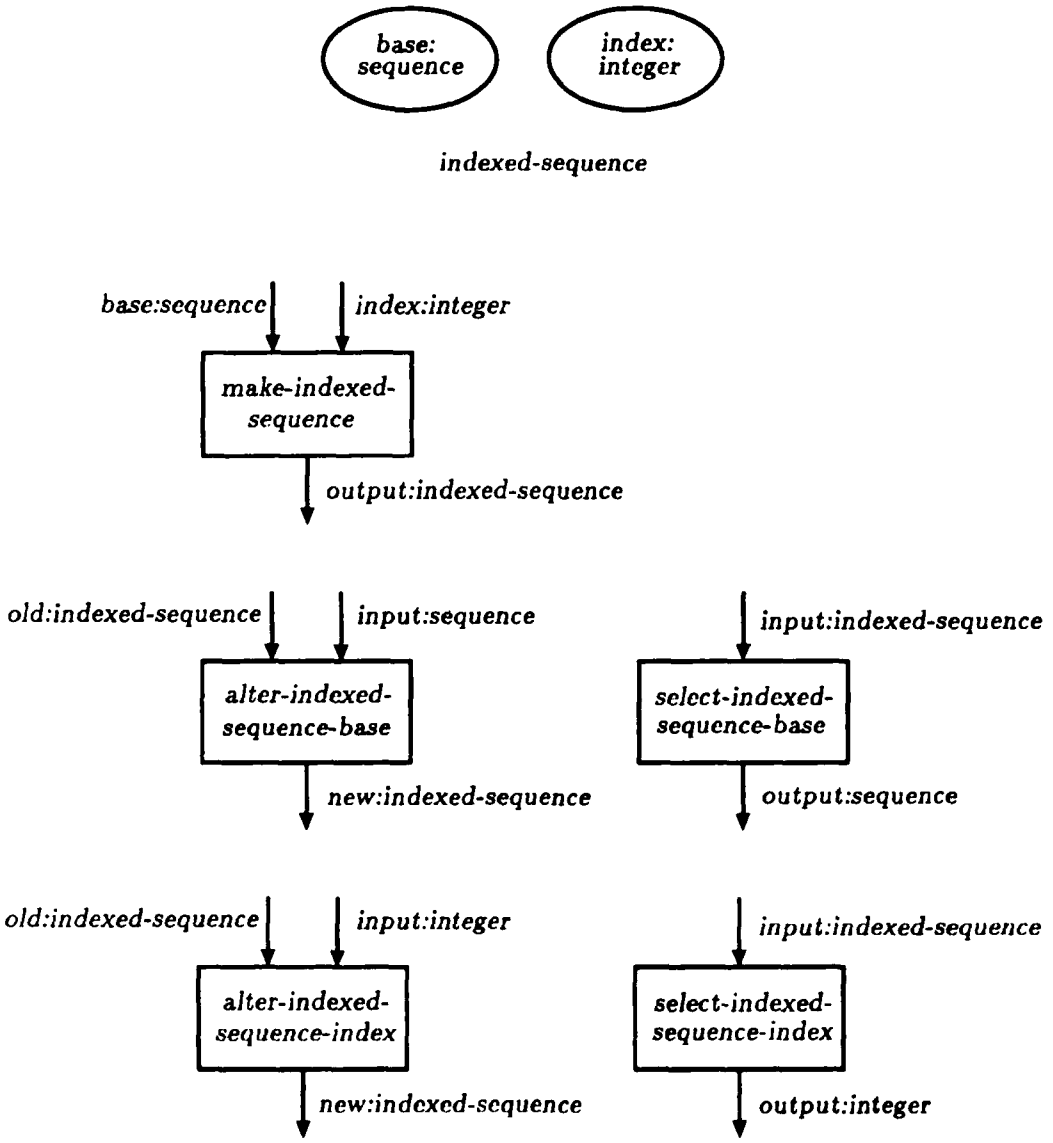
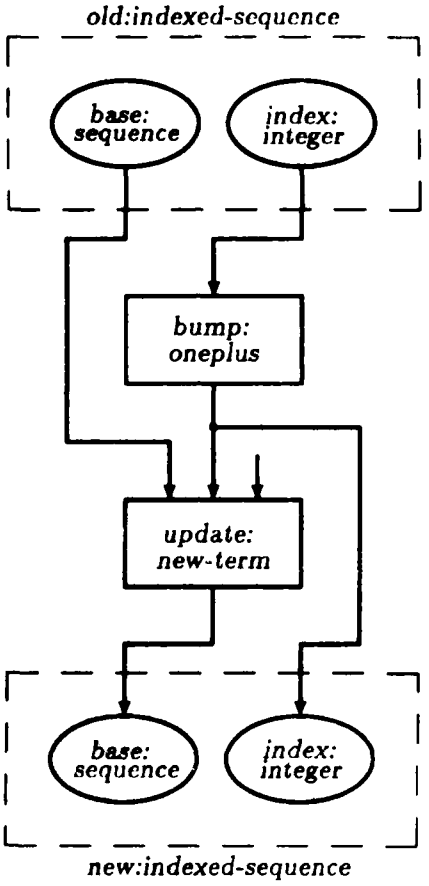
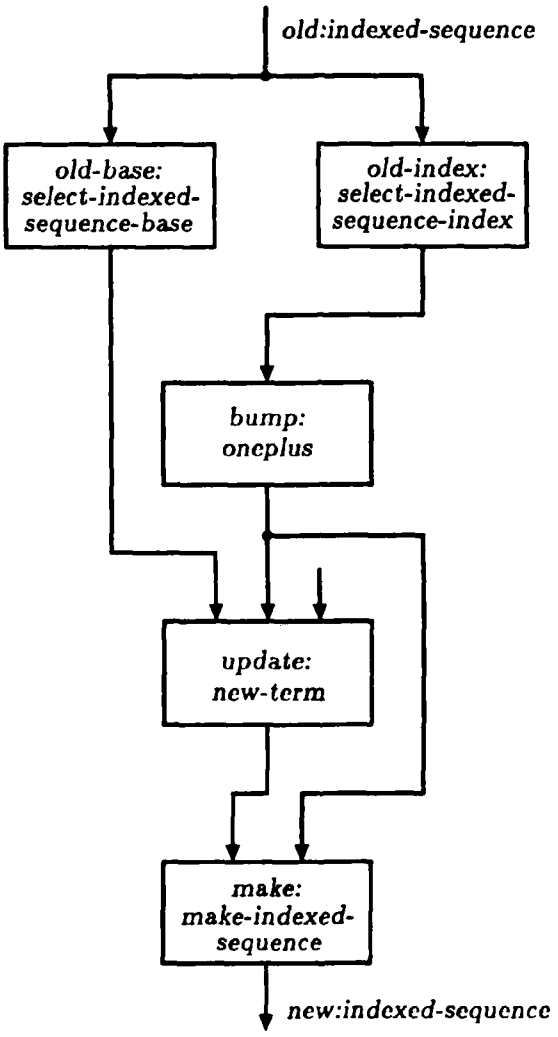


Figure 23. An example of a data plan and the corresponding accessors. The accessors are implicitly defined as part of the definition of the data plan.



bump-and-update

Figure 24. An example of a hierarchical plan with a mixture of data and computation roles. The plan Bump-and-Update captures the clichéd pattern of operations on an indexed sequence in which the index is incremented (Bump) and a new term is stored (Update).



bump-and-update

Figure 25. An equivalent version of Bump-and-Update (see Figure 24), in which explicit accessors have been used instead of using data plans.

Data Plans

The type of a role in a plan can also be a primitive data type, such as Integer, Sequence, or Set. A plan all of whose roles are data types (or hierarchically, data plans) is called a *data plan*. Data plans are used to represent standard data structure aggregations which appear in the implementation of more abstract data types. For example, Figure 23 shows the data plan Indexed-Sequence, which represents the common cliché of a sequence (Base) with an associated index pointer (Index). The Base is typically implemented more concretely as an array. This data plan is, for instance, part of many implementations of buffers, queues, and stacks.

The logical portion of a data plan associates an *invariant* with the data aggregation. For example, the invariant of Indexed-Sequence states that the Index must be greater than or equal to zero and less than or equal to the length of the Base.

The definition of a data plan, such as Indexed-Sequence, automatically defines a corresponding collection of input/output specifications for the standard data structure accessors:

- A *constructor*, which takes an instance of the appropriate type for each of the roles, and produces a new instance of the data plan with those parts. A precondition of this operation is that the inputs satisfy the invariant of the data plan.
- A *selector* for each role, which takes an instance of the data plan, and returns the corresponding part.
- An *alterant* for each role, which take an instance of the data plan and an instance of the appropriate type for the role, and destructively modifies the instance of the data plan by replacing the corresponding part with the new part. A precondition of this operation is that the new part together with the old parts for the other roles satisfy the invariant of the data plan.

The naming conventions for these accessors, their inputs, and their outputs, are illustrated in Figure 23.

Implicit Accessors

In general, a hierarchical plan may have a mixture of data and computation roles. Figure 24 shows an example of a hierarchical plan, Bump-and-Update,

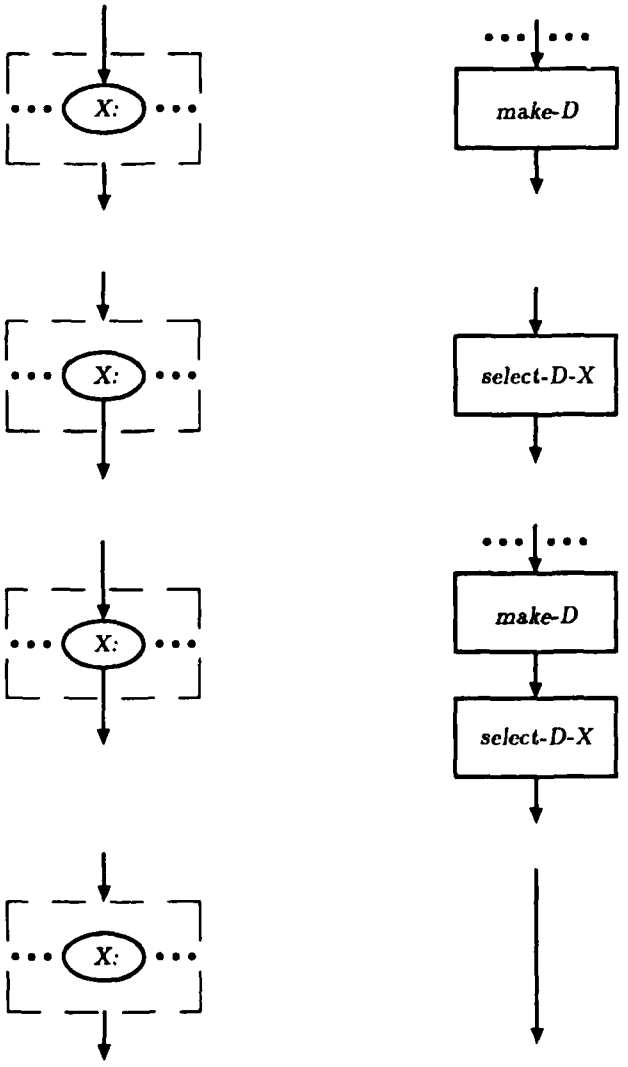


Figure 26. For a data plan, D, with roles ..., X, ..., this figure shows how each possible arrangement of data flow to a single role (on the left) is translated into explicit accessors (on the right). See Figure 27 for more general case.

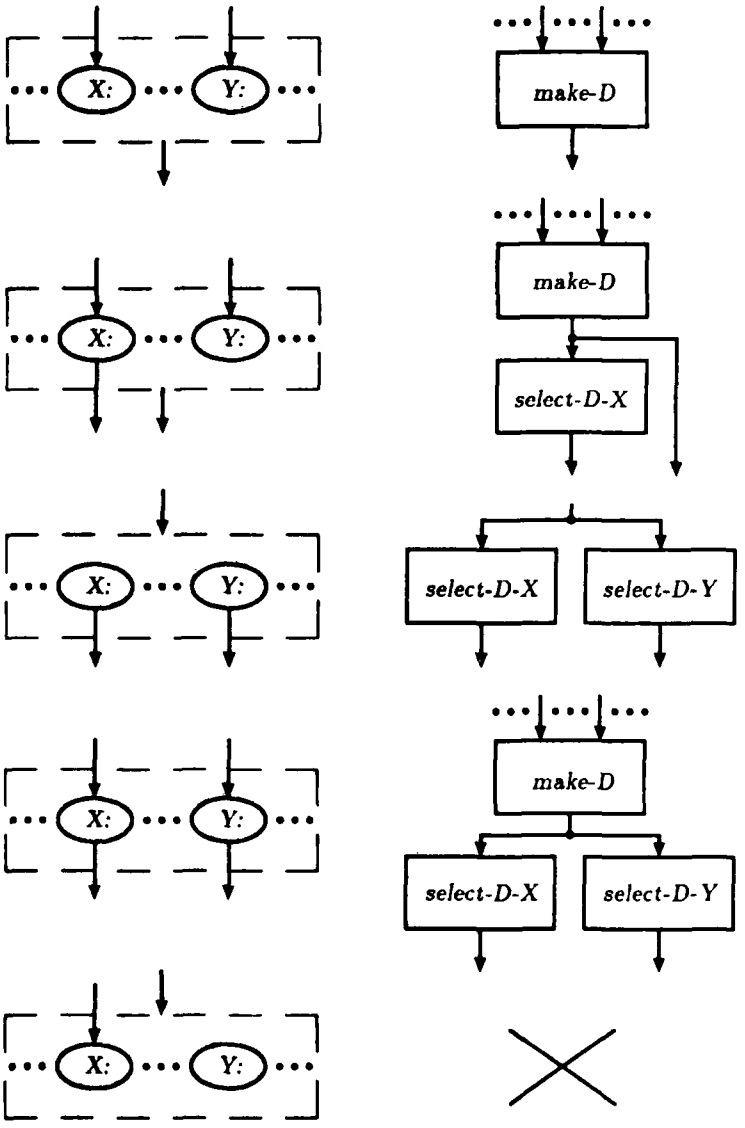


Figure 27. For a data plan, D, with roles ..., X, ..., Y, ..., this figure shows how various combinations of data flow to multiple roles (on the left) is translated into explicit accessors (on the right). See Figure 26 for simpler cases.

which has the data plan Indexed-Sequence as a subplan (twice). This plan expresses the clichéd pattern of operations on an indexed sequence in which the index is incremented and a new term is stored at that location in the sequence, as for example, in the following code:

```
(DEFSTRUCT INDEXED-SEQUENCE BASE INDEX)
...
(LET ((I (1+ (INDEXED-SEQUENCE-INDEX Q)))
      (S (COPY-SEQ (INDEXED-SEQUENCE-BASE Q))))
  (SETF (ELT S I) ITEM)
  (MAKE-INDEXED-SEQUENCE :BASE S :INDEX I))
```

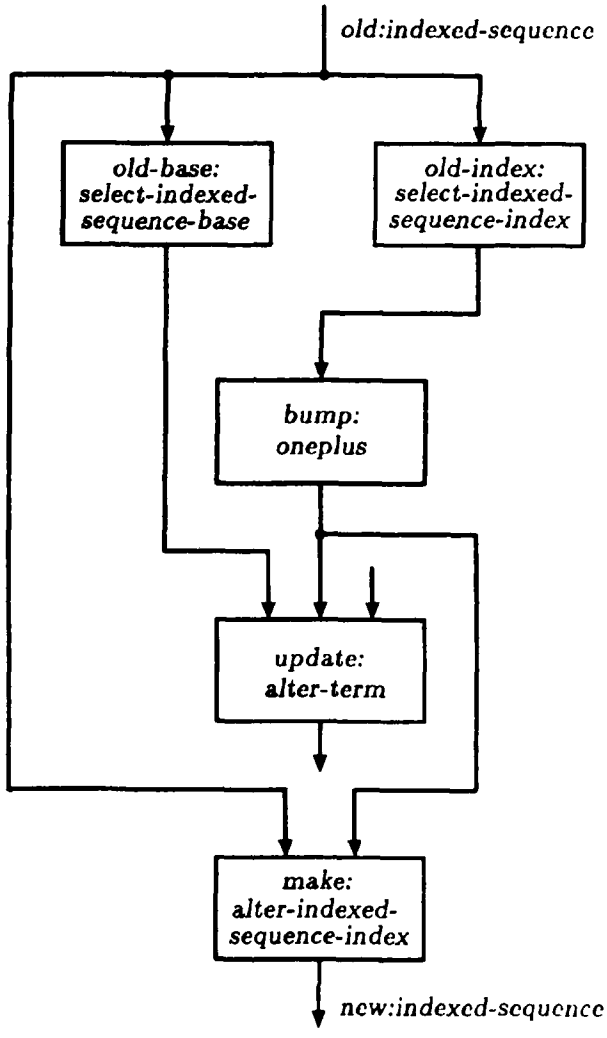
Notice that the Bump-and-Update plan is purely functional, i.e., there are no side effects. New-Term (the type of the Update step) is a predefined input/output specification associated with the primitive data type Sequence—it returns a *copy* of the input sequence, with one term changed. Since there is no sequence primitive in Lisp corresponding to New-Term, the code above uses a combination of `COPY-SEQ` and `SETF` of `ELT` to implement this operation. A related version of this plan which uses side effects is discussed below.

Notice also that the selector and constructor operations in the code above for Bump-and-Update do not appear explicitly as boxes in the plan diagram. It is a convenient feature of plan diagrams that these accessors are implicit in the way data flow is connected to the roles of a data plan. For example, the version of Bump-and-Update in Figure 24 can be taken as an abbreviation for the version in Figure 25, in which the accessors are made explicit. The general rules for interpreting data flow involving data plans are illustrated in Figures 26 and 27.

3.6 Side Effects

Side effects are modelled in the Plan Calculus by introducing input/output specifications which destructively modify their inputs. For example, the destructive version of New-Term, called Alter-Term, has the same input and output roles as New-Term. Its postconditions, however, specify that the Old sequence is destructively modified to obtain the New sequence. (The formal statement of this condition involves using a situational calculus for modelling mutable objects—see [41, 42, 40].)

Figure 28 shows an example of a plan, called Destructive-Bump-and-Update, involving side effects. This plan is the more common, destructive



destructive-bump-and-update

Figure 28. The destructive version of Bump-and-Update.

version of Bump-and-Update, corresponding to the code below. (Cross-referencing between the destructive and non-destructive versions of specifications and plans is part of the library structure [39].)

```
(LET ((I (1+ (INDEXED-SEQUENCE-INDEX Q))))
  (SETF (ELT (INDEXED-SEQUENCE-BASE Q) I) ITEM)
  (SETF (INDEXED-SEQUENCE-INDEX Q) I))
```

Notice that the plan diagram for Destructive-Bump-and-Update has explicit accessors, such as Alter-Indexed-Sequence-Index, for the parts of the indexed sequence. The abbreviated data flow notation for data plans described above cannot be used in plans with side effects because the correct expansion of the abbreviations in the presence of side effects requires nonlocal reasoning. For example, in Destructive-Bump-and-Update, there is no alterant for the Base of the indexed sequence, because the destructive modification of the sequence in the Update (Alter-Term) step also achieves a destructive modification of the whole indexed sequence of which it is a part.

In the Plan Calculus, side effects arise only in connection with the destructive modification of arrays, records, and other mutable data structures. Most of the side effects in conventional programming languages, namely assignment statements, are replaced by the use of data flow in the Plan Calculus. (An exception is the use of global variables, whose current value *is* best thought of as part of the state of the system. These are modelled using the primitive mutable data plan, Cell, which has a single role, called Contents.)

In general, reasoning about side effects can be quite complex, especially if mutable objects may overlap (see [53, 54]).

3.7 Recursively Defined Plans

Hierarchical plans can be recursively defined, i.e., the type of one or more of the subplans can be the same as the type of the plan. For example, Figure 29 shows the recursive data plan defining the standard list and binary tree abstractions.

Recursive computations are also represented using recursive plan definitions. For example, Figure 30 shows the recursively defined plan, called Bintree-Enumeration, for enumerating (visiting every node of) a binary tree. In the usual Lisp implementation of binary trees as cons cells, the following code is an implementation of this plan.

```
(DEFUN ENUMERATE (TREE)
  ...
  (UNLESS (ATOM TREE)
    (ENUMERATE (CAR TREE))
    (ENUMERATE (CDR TREE))))
```

Notice, however, that this code makes an ordering commitment that is not required by the Bintree-Enumeration plan. In this code, the nodes of the tree are walked in left-to-right order (assuming CAR corresponds to Left and CDR to right). The Bintree-Enumeration plan is more general—it does not force the traversal to occur in any particular order. An advantage of the Plan Calculus over conventional program text is that it allows the expression of more general clichés, such as this. Furthermore, to constrain the Bintree-Generation plan to the traversal order used in the code above, all that is required is to add a control flow arc from Continue-Left.End to Continue-Right.Exit.

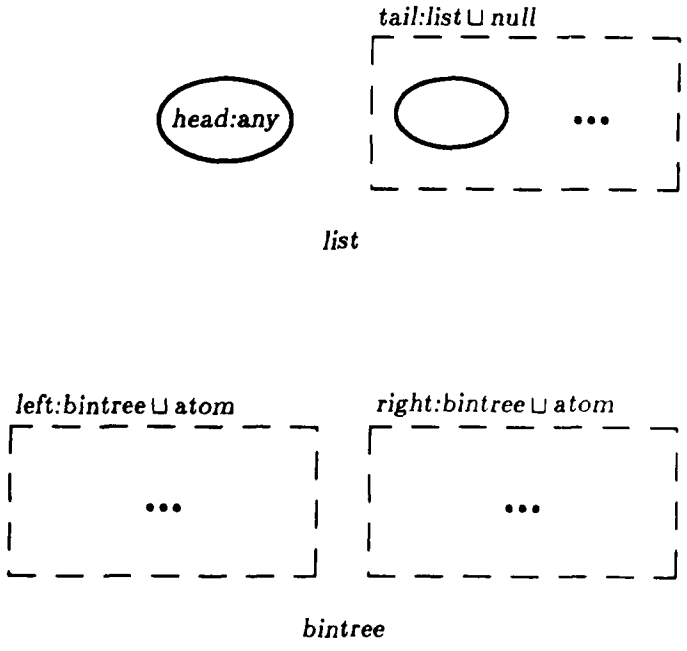


Figure 29. Two examples of recursively defined data plans. Note the use of disjunctive types.

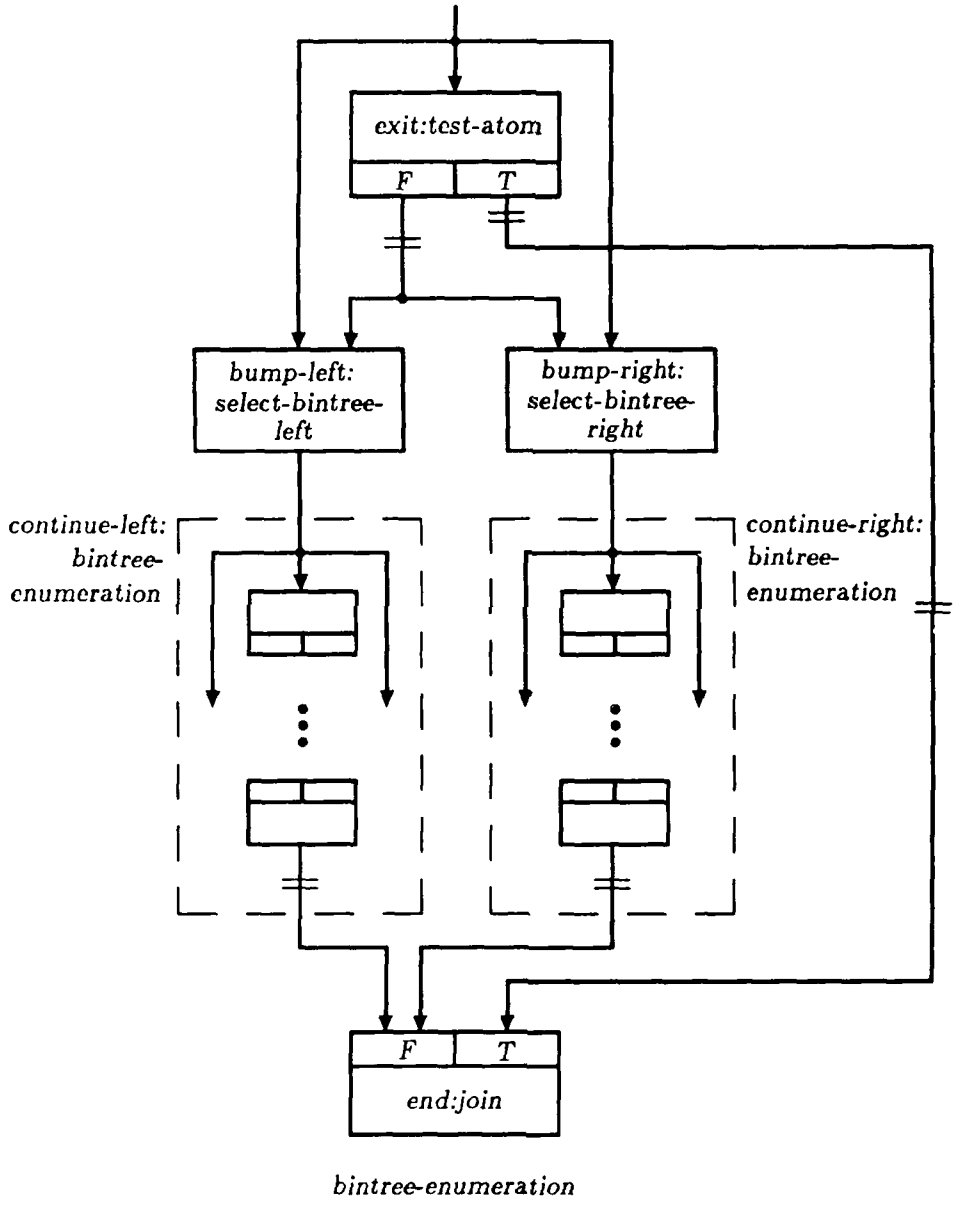


Figure 30. The recursive definition of the plan for enumerating a binary tree.

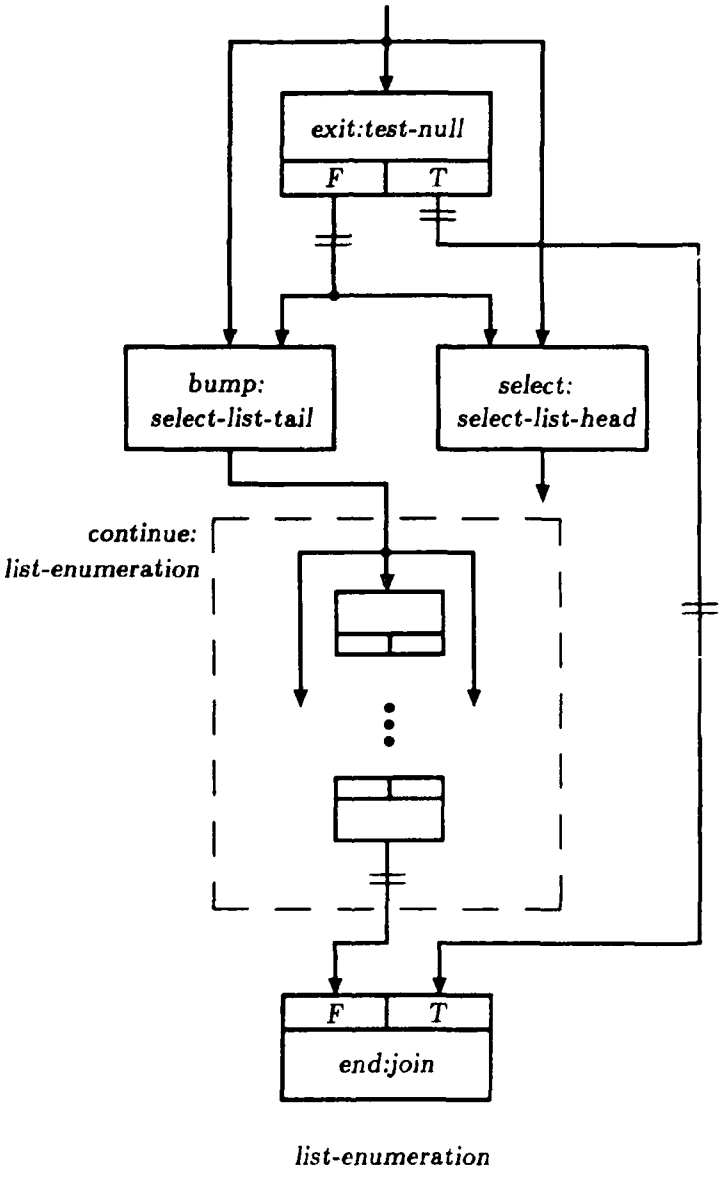


Figure 31. Iterative (tail-recursive) plan for enumerating a list.

Iterative Computations

Iterative computations are represented in the Plan Calculus by recursively defined plans. For example, Figure 31 shows the plan for enumerating the elements of a list. In the standard implementation of lists in Lisp, the following code is an implementation of this plan.

```
(LOOP
  (IF (NULL L) (RETURN))
  ... (CAR L) ...
  (SETQ L (CDR L)))
```

This is the familiar CAR, CDR, NULL cliché that appears in several different syntactic forms in the hash table example of Section 2. This cliché can alternatively be coded in the following recursive form, which mirrors more closely the structure of the plan in Figure 31.

```
(DEFUN ENUMERATE (L)
  (WHEN L
    ... (CAR L) ...
    (ENUMERATE (CDR L))))
```

The two versions of the code above are computationally equivalent. In both cases, the amount of memory used in the computation does not need to grow with each repetition of the body. (It is a defect of some compilers and interpreters that these two versions are not executed in the same way.) A recursive definition that corresponds to an iterative computation is often referred to as *tail-recursive*. Although iterative computations are often loosely referred to as "loops", the essential characteristic of iteration is not the existence of a cycle in control flow, but rather, the fixed space requirements of the computation.[7]

The difference between a singly-recursive plan that gives rise to an iterative computation, and one that gives rise to a recursive computation has to do with whether there are any operations to be performed "on the way up", i.e., after the recursive invocation. This point can be illustrated by comparing plans for the recursive versus iterative computation of factorial.

A plan for the recursive computation of factorial is shown in Figure 32. This plan corresponds to the following code.

[7] For a further discussion of the relationship between iteration, recursive definition, and looping constructs, see [1], pp. 32-33.

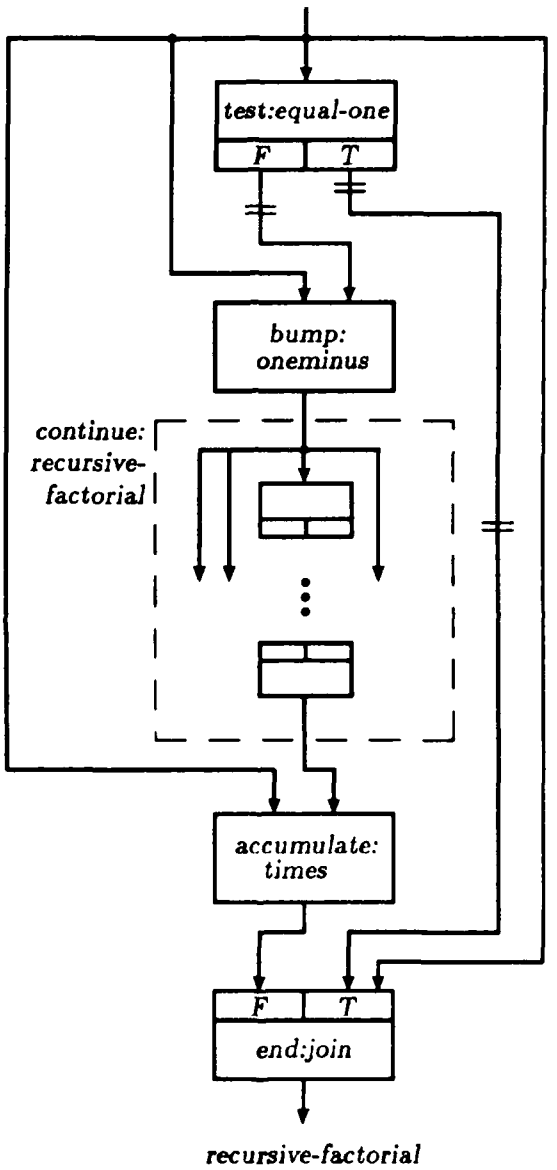


Figure 32. Linear recursive plan for the computation of factorial.

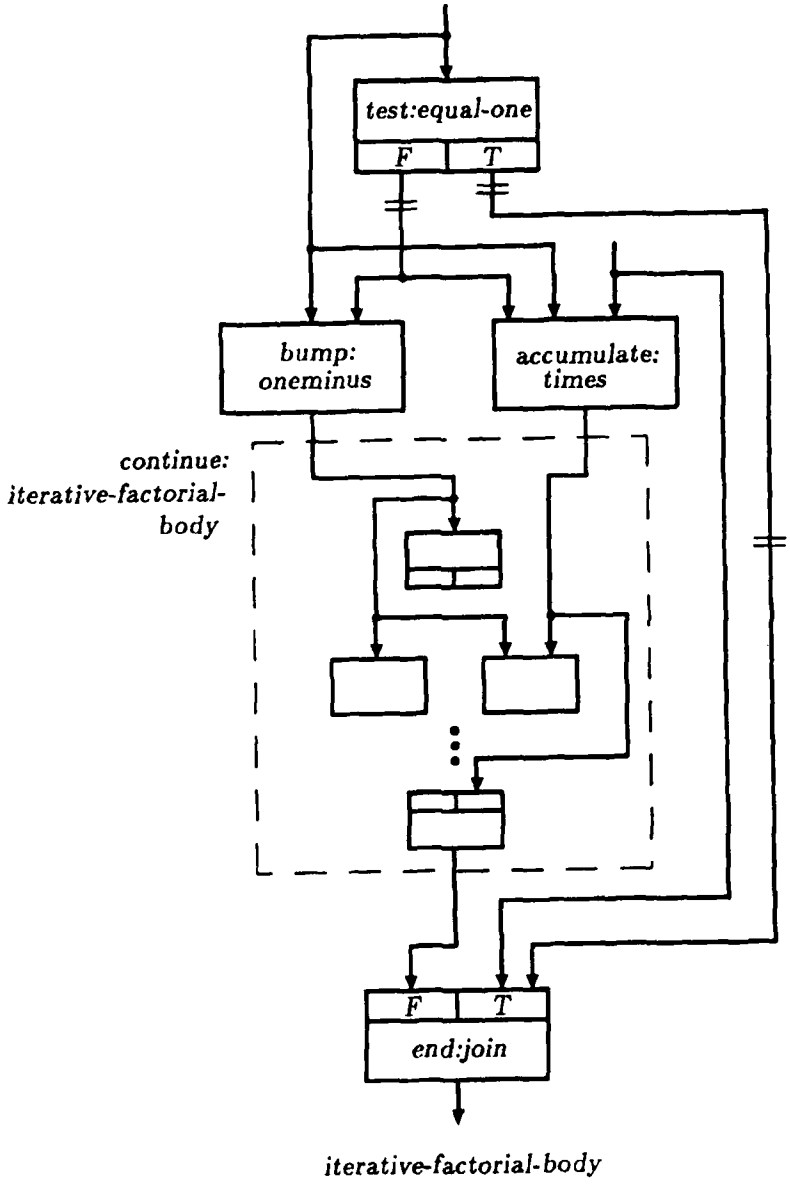


Figure 33. Iterative (tail-recursive) plan for the computation of factorial. Note that an auxiliary plan definition (not shown here) is required to specify initialization of the accumulated product to 1.

```
(DEFUN FACT (N)
  (IF (= N 1)
      1
      (* N (FACT (1- N)))))
```

Note that the multiplication (Accumulate) step in this plan requires input from the end of the recursive invocation, and therefore must come after the recursion. This computation is not iterative, but linear recursive—memory grows linearly with the number of repetitions of the body.

A tail-recursive plan for the iterative computation of factorial is shown in Figure 33. This plan corresponds to the following recursive definition.

```
(DEFUN FACT-ITER (N F)
  (IF (= N 1)
      F
      (FACT-ITER (1- N) (* N F))))
```

Factorial of n is computed by calling `FACT-ITER` with the accumulated product (`F`) initialized to 1.

```
(DEFUN FACT (N)
  (FACT-ITER N 1))
```

This can alternatively be coded as a loop, as follows.

```
(DEFUN FACT (N)
  (LET ((F 1))
    (LOOP
      (IF (= N 1) (RETURN F))
      (SETQ N (1- N))
      (SETQ F (* N F)))))
```

Notice that in the plan in Figure 33 there are no computations to be performed after the recursive invocations. (Joins do not count as computations, but are really part of the data and control flow constraints.)

Another example of a tail-recursive plan is the Linear-Search plan in Figure 11, which captures the linear search cliché used in the hash table example. A taxonomy of iterative clichés has been developed by Waters [59] and elaborated by Rich [39].

3.8 Overlays

Programming knowledge includes understanding many kinds of relationships between plans. One important kind of relationship is how an instance of one plan can be viewed as an instance of another. Overlays are the general facility in the Plan Calculus for representing such shifts of viewpoint. Examples of overlays given below capture the common programming notions of implementing a specification, data abstraction, and optimization.

Implementing a Specification

Figure 34 is an example of a simple overlay representing implementation knowledge. The right side of the diagram is the Absolute-Value input/output specification. The left side of the diagram is the plan, Compute-Absolute-Value, which tests whether a number is negative and, if so, negates it. This overlay represents the fact that the Compute-Absolute-Value plan is a correct implementation of the Absolute-Value specification. (A statement of the correctness conditions is given below.) Notice the distinction being made here between the specification for absolute value, and one way of computing it, even though these two are very close in this example. Although Compute-Absolute-Value is the most obvious way of implementing Absolute-Value, there are other possible ways—for example, squaring the number and then taking the square root. Each way of implementing Absolute-Value is represented by a different overlay, all of which have the same right side.

In addition to a left and right side, an overlay diagram also includes a set of hooked lines, called *correspondences*, which identify the corresponding objects in the two points of view.[8] In Figure 34, for example, the correspondences identify the input of the absolute value specification with the input of the test of the implementation plan, and the output of the absolute value specification with the output of the join of the implementation plan.

Formally, an overlay defines a mapping from the set of instances of the left side plan (the domain) to the set of instances of the right side plan (the range). There may be different overlays with the same domain and/or range. In order to be correct, the mapping defined by an overlay must be single-valued, total and onto.[9]

[8] The idea of correspondences was stimulated in part by Sussman's "slices" [57], which he used to represent equivalences between electronic circuits.

[9] A mapping is *onto* iff each element of the range is the image of some element of the

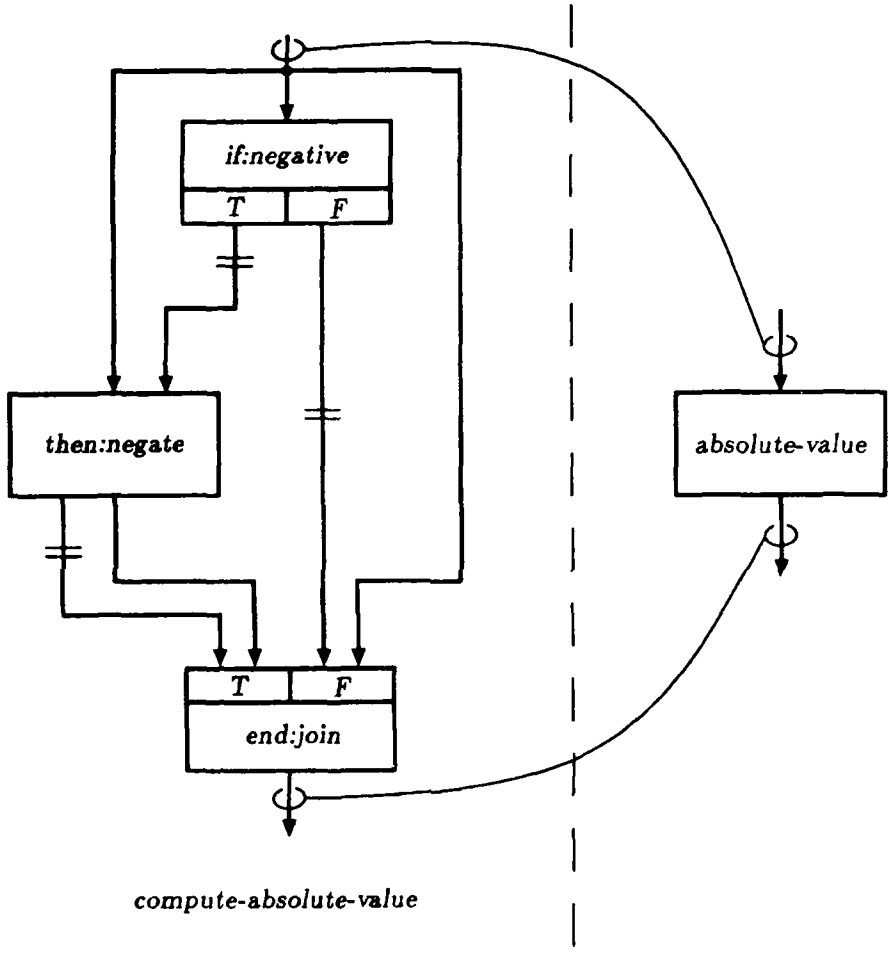


Figure 34. Overlay for an implementation of absolute value by testing and negating.

The single-valued condition guarantees that the implementation process loses no information, i.e., for a given overlay, the specification can always be recovered from the implementation. The mapping may, however, be many-to-one, so that the implementation typically is not uniquely determined by the specification.

The total condition guarantees that each implementation instance corresponds to some specification. (Typically, this is achieved by restricting the domain of the overlay until this condition is satisfied.)

Finally, the onto condition guarantees that each specification is implementable.

The logical sublanguage and the formal semantics of the Plan Calculus provide the basis, in principle, to formally verify all of the overlays described in this paper. An automated proof system which can be used for this task has been implemented by Feldman and Rich [44, 15]. Thus far, however, these conditions have only been used as an intuitive guide to writing overlays.

Using Overlays in Analysis and Synthesis

The knowledge encoded in an overlay can be used in both analysis and synthesis of programs. In analysis by inspection, the left side of an overlay is matched against the plan representation of the program under analysis. If a match is found, then the part of the plan matching the left side of the overlay can be replaced by the right side of the overlay. The correspondences provide the information needed to connect the right side of the overlay with the appropriate parts of the surrounding plan. (See example in Figure 35.)

The repeated application of this recognition process can be thought of as a kind of parsing, where each overlay defines a grammar rule. (The sides are reversed: The right side of the overlay corresponds to the reduced side of the grammar rule; the left side of the overlay corresponds to the expansion of the rule.) Note that this grammar will typically be ambiguous,[10] because there may be several overlays with the same left side, and also because the parts of a plan may often be grouped in several different ways. Wills [63] has constructed an automated system which performs analysis by inspection using a graph-parsing approach.

---

domain.

[10] A grammar is ambiguous iff some sentences in the language do not have a unique derivation.

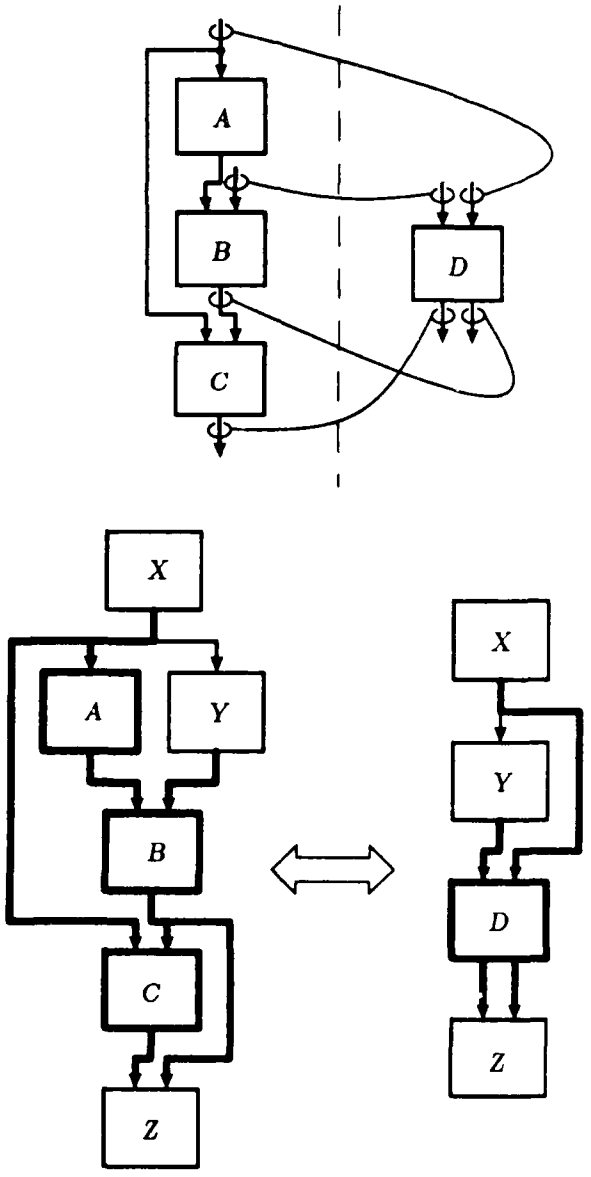


Figure 35. The example overlay at the top of the figure is shown being used in analysis and synthesis. In analysis by inspection, the left side of the overlay is recognized in the larger plan at the lower left of the figure. The part of the larger plan matching the left side of the overlay is highlighted in bold. It is replaced by the right side of the overlay as shown. In synthesis by inspection, this process is reversed.

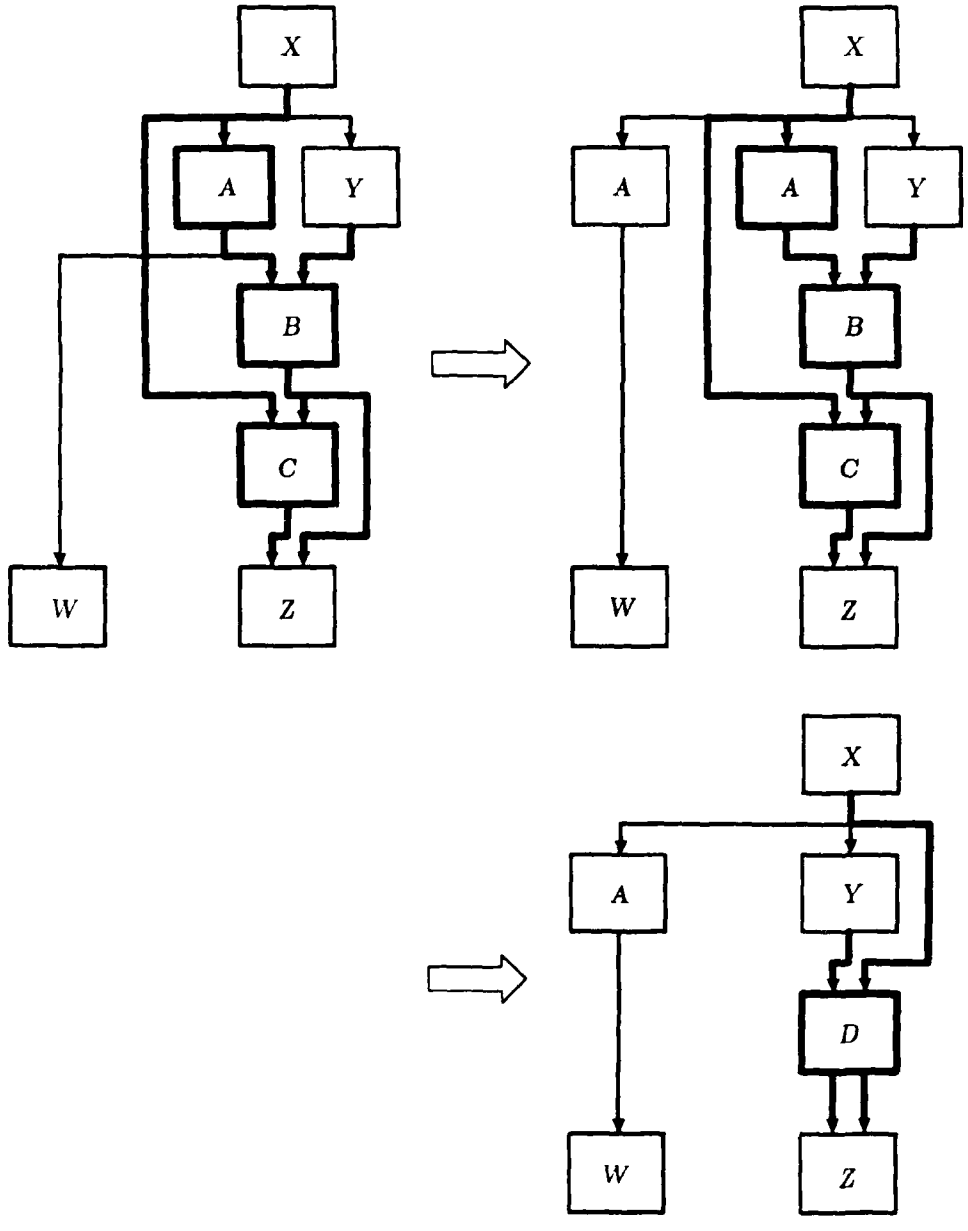


Figure 36. An example of matching a plan in which copying is required before replacement. The part of the larger plan matching the left side of the overlay in Figure 35 is highlighted in bold. Notice that *A* is copied first, and then the matched part of the plan is replaced by the right side of the overlay.

In synthesis by inspection, the right side of an overlay is matched against the plan representation of the current synthesis state. If the right side of the overlay is a single box, as in the case of implementation overlays, then this is trivial. We will see below that the right side can also be a plan. The part of the plan matching the right side of the overlay is replaced by the left side of the overlay, again using the correspondences to get the right connections. (See Figure 35.) In the grammar metaphor, synthesis by inspection corresponds to running the same grammar as a generator. A system which supports a kind of synthesis by inspection has been implemented by Waters [61].

Note that in the process of matching and replacement, parts of the matched plan may need to be copied before replacement is made. The parts of the matched plan that need to be copied are any operations or tests whose output has data flow going outside the matched area, and for which there is no corresponding output on the other side of the overlay. Figure 36 shows an example of when copying is required in the use of an overlay in analysis by inspection. The same copying would be required in the synthesis direction if the same plan were the right side of another overlay.

Data Abstraction

Data abstraction is represented in the Plan Calculus by overlays between data plans. The data plan on the left side of the overlay is what is typically called the concrete (or implementation, or representation) data type; the data plan on the right side of the overlay is the abstract data type. As with overlays in general, a data overlay must define a single-valued, total and onto mapping from instances of the concrete data type to instances of the abstract data type. This mapping is typically called the *abstraction function* in the data abstraction literature (e.g., [30]).

Only the domain and range types of a data overlay can be indicated in plan diagrams. The definition of the abstraction function requires the logical/mathematical sublanguage. For example, Figure 37 shows the data overlay, Indexed-Sequence-as-List, which represents one way of implementing a list using an indexed sequence. The abstraction function for Indexed-Sequence-as-List is defined as follows: The head of the list corresponds to the term of the base sequence indexed by the index. The tail of the list is recursively defined as the list implemented by the indexed sequence with same sequence and one minus the index. The empty list (nil) corresponds to

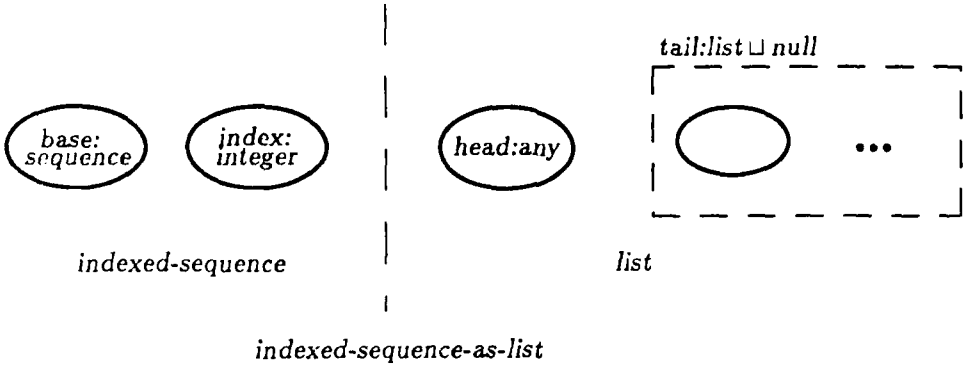


Figure 37. An example of implementation knowledge involving data abstraction. The data overlay, Indexed-Sequence-as-List, specifies how to implement a list using a sequence and an index. Only the domain and range are indicated in the plan diagram.

the indexed sequence with index zero.

Data overlays are typically used to define other overlays. For example, Figure 38 shows the definition of an overlay which describes how to implement the Push[11] operation on a list, when the list is implemented as an indexed sequence (according to Indexed-Sequence-as-List). The left side of this overlay is the Bump-and-Update plan introduced earlier. In this implementation, the Old and New indexed sequences of the Bump-and-Update plan correspond to the Old and New lists of Push, respectively.[12] The object which becomes the new term in Bump-and-Update corresponds to the object being pushed onto the list.

Notice that two of the correspondences in the diagram for Bump-and-Update-as-Push in Figure 38 are annotated with the name of the data overlay Indexed-Sequence-as-List. This means that the Old indexed sequence of Bump-and-Update *viewed as a list according to Indexed-Sequence-as-List* corresponds to the Old input of Push, and similarly for the New roles. This

[11] The postconditions of Push state that the head of the New list is equal to the Input, and the tail of the New list is equal to the Old list.

[12] Recall that the plan diagram shown for Bump-and-Update in Figure 38 is actually an abbreviation for the version with explict accessors shown in Figure 25. With explicit accessors on the left side of overlay, the correspondence involving the Old indexed sequence would connect to the input to the selectors at the top of the plan; the correspondence involving the New indexed sequence would connect to output of the constructor at the bottom.

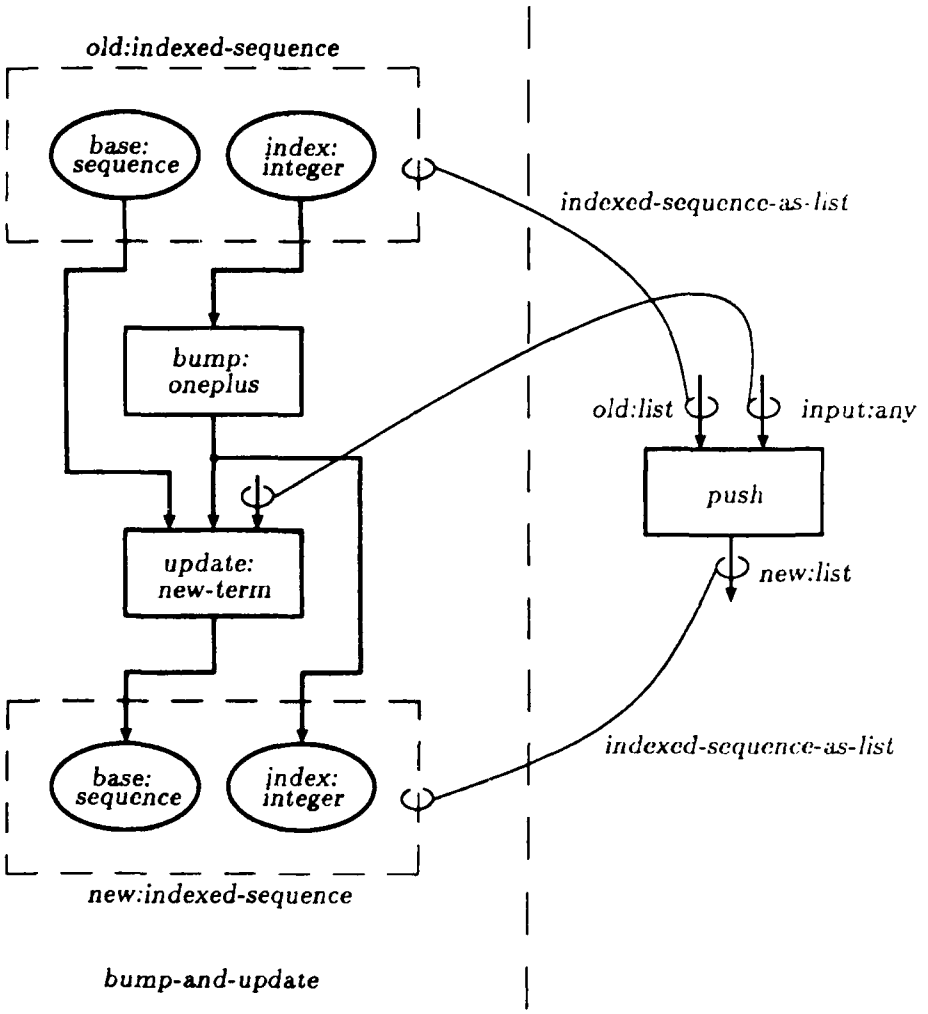


Figure 38. The Bump-and-Update-as-Push overlay specifies how to implement the Push operation using the data abstraction Indexed-Sequence-as-List (see Figure 37).

labelling convention is quite general. Any correspondence can be labell with the name of any function having the appropriate domain and rang This means that this function is applied to the object on the left to ol tain the corresponding object on the right. One can think of an unlabell correspondence as meaning the identity function.

Notice that using data overlays, the same data abstraction can be impl mented differently in different contexts; this is awkward in some prograi ming languages.

Finally, notice that the implementation knowledge in Figure 38 is f the most abstract case, namely an unbounded list implemented using a unbounded sequence, without side effects (the input and output lists of Pu are not identical; New-term is the non-destructive operation on sequence A plan library would also include overlays between versions of these plans which the Push operation can cause overflow, the base sequence has a fix length, and various operations are destructive.

Optimization

The most general form of overlay has a non-atomic plan diagram on ea side. Such overlays are most often used to capture optimization knowled For example, Figure 39 shows an overlay having to do with optimizing certain pattern of operations on a list. The right side of this overlay is a pl in which an object is pushed onto a list, the list is sorted, another obj is pushed onto the sorted list, and then it is sorted again. This pattern operations can be optimized as shown by the plan on the left side of t overlay, in which the first sorting operation is omitted. One can think of t overlay as embodying a small lemma in the theory of lists and sorting.

One would not particularly expect a programmer to write code mat ing the right side of this overlay. However, patterns requiring optimizati can easily arise in the process of automated synthesis, when higher le operations are expanded into implementations. For example, a simple imp mentation for adding an object to a sorted list is to push the object onto t list and then sort. Two such operations on the same sorted list implement this way would give rise to the pattern on the right side of this overlay.

Using an overlay such as Figure 39 in the synthesis direction, i.e., mat ing the right side and replacing it by the left side, amounts to applying optimization. Using an overlay such as Figure 39 in the analysis directi i.e., matching the left side and replacing it by the right side, amounts

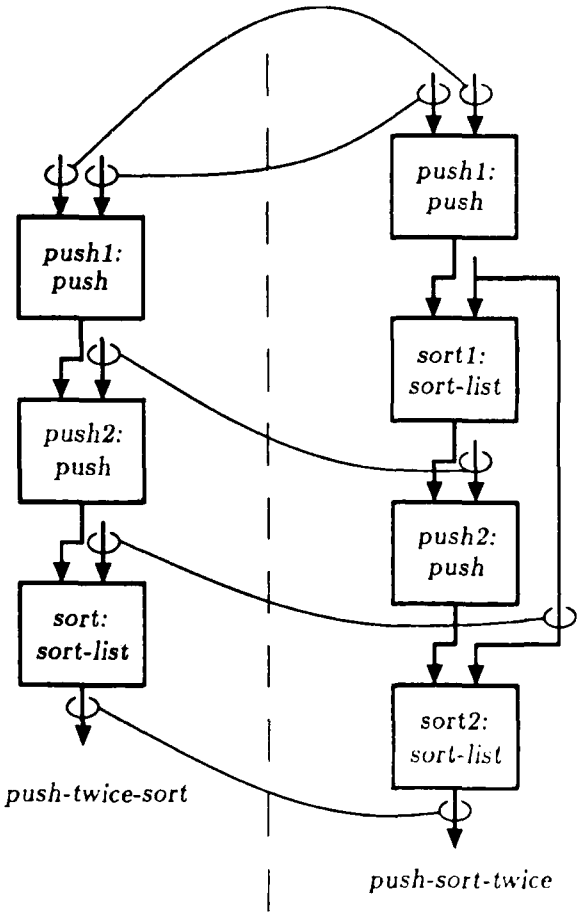


Figure 39. An example of an overlay encoding optimization knowledge. The right side of the overlay is the unoptimized form; the left side is the optimized form. The Sort-List specification takes as input a list and an order predicate. Its output is a list with the same elements as the input list, sorted according to the order predicate.

"undoing" an optimization. It is often necessary to undo optimizations in order to facilitate further recognition.

In the grammar metaphor, an overlay with non-atomic plans on both the left and right side corresponds to a context-sensitive grammar rule. Undoing optimizations as part of recognition is therefore an inherently expensive process.

3.9 Summary

This section summarizes the structural sublanguage of the Plan Calculus with a formal definition of its syntax. Note that the syntax of plan diagrams allows plans that are not semantically well-formed, for example, for which no possible executions exist (see [40, 41, 42] for more on semantics).

We begin with a set of *primitive types*, which are in the language. These types provide the primitive data vocabulary, such as Integer, Sequence, and Set, out of which specifications are built. The primitive type Situation is used to model control flow and side effects.

There are two kinds of composite structures in the language: *specifications* and *overlays*.

A *specification* is composed of a labelled tuple and a set of labelled *edges*. A labelled tuple is an tuple in which the components are selected by arbitrary distinct symbols (labels) instead of numbers. The set of valid labels for the components of a specification are called its *roles*. The components of a specification are either specifications or primitive types.

The edges of a specification are pairs of *paths* in the specification. A path in a specification, A, is defined recursively as follows:

> If r is a role of A, then r is a path in A.
>
> If B is the component of A selected by r, and p is a path in B, then $r.p$ is a path in A.

Given these definitions, the terminology of plan diagrams introduced in the preceding sections arises out of classifying specifications according to their components, as follows.

An *input/output specification* is a specification with exactly two Situation components. These are the entry point and exit points roles, which are labelled by convention In and Out. The remaining roles are partitioned into

two disjoint subsets, called the *inputs* and the *outputs*. There are no edges in an input/output specification.

A *test specification* is a specification with exactly three Situation components. These are the entry point, and the success and failure exit points, which are labelled by convention In, Succeed, and Fail, respectively. The remaining roles are partitioned into three disjoint subsets, called the *inputs*, the *success outputs* and the *failure outputs*. There are no edges in a test specification.

A *data plan* is a specification, all of whose components are either primitive data types (i.e., primitive types other than Situation) or data plans. There are no edges in a data plan.

A *plan* (the general case) is a specification, all of whose components are either input/output specifications, test specifications, primitive data types, or plans. The edges in a plan are labelled to indicate whether they are control flow or data flow. Data plans are a special case of plans. The term *temporal plan* is sometimes used to distinguish plans which are not data plans, i.e., which include at least one input/output or test specification.

An *overlay* is composed of a pair of specifications and a set of labelled edges. The edges of an overlay are pairs of paths, in which the first element of each pair is a path in the first specification and the second element of each pair is a path in the second specification. The edges of the overlay are called *correspondences*, and are labelled with the name of the function used to map objects from the left side to the right side of the overlay.

4 Conclusion

This section discusses the relationships between the Plan Calculus and other formalisms, reviews some of the limitations of the Plan Calculus, and summarizes further work to be done.

4.1 Relation to Programming Languages

An often asked question is: Is the Plan Calculus (just) another (very high level) programming language? As with many such questions, the heart of the answer lies in defining the terms. In this case, it depends just what is meant by "programming language." Modern programming languages have two essential purposes:

- To describe computations precisely enough to be executed by a machine.
- To serve as a communication medium between program writers and *human readers.*

In contrast, the two essential purposes of the Plan Calculus are:

- To describe programming clichés in a canonical, easy to combine, and language-independent form.
- To serve as a medium for automated manipulation of programs.

As we will see, these respective purposes are in some ways compatible, and in other ways conflicting. The answer to the question is therefore not a simple yes or no.

Conventional programming languages force the programmer to provide enough detail so that a simple local interpreter (e.g., hardware, perhaps with an intermediate compilation step) can execute the code. Unfortunately, much of this detail, such as the variety of special forms used for binding variables, looping, conditional branching, etc., is often irrelevant to respect to the algorithmic content of the code. As discussed in Section 3.1, this aspect of conventional programming languages conflicts with the canonicalness goal of the Plan Calculus.

The goals of serving as a human communication medium and serving as a medium for automated manipulation can also conflict. For human communication, a critical restriction is the fact that information must ultimately

be laid out on a two-dimensional structure (i.e., on the retina). In contrast automated manipulation systems have no such inherent topological restriction. It is possible (and often desirable) in such systems to have very highly interconnected information structures in which many kinds of information are localized at a single point.

As discussed in Section 3.1, the graphical nature of the Plan Calculus is motivated by a desire for ease of manipulation by an automated tool. As plan diagrams grow in size, they very quickly become hard for humans to understand visually. Although it may turn out that the Plan Calculus is a good starting point for a graphically-oriented human communication environment, how to best use graphics for programming is still an open research question.

Wide-Spectrum Languages

Recently, the notion of programming language has been extended to include so-called very high level languages (VHLL's). Some of these VHLL's are executable, although not by a simple local interpreter, and not very efficiently. Others are really specification languages, in the sense that the compiler is making significant implementation decisions, such as the choice of data structures and algorithm. Furthermore, most VHLL's are also *wide spectrum*, i.e., they include a conventional high-level language as a sublanguage.

The Plan Calculus is also a wide-spectrum language. The input/output and test specifications used in a given plan may correspond to operations typically available in a conventional programming language, or they may be much more abstract. To illustrate this point, consider how one translates a program from a conventional high-level programming language into the Plan Calculus. First, the primitives of the programming language are divided into two categories:

- The "connective tissue" primitives, such as `PROG`, `COND`, `SETQ`, `GO`, and `RETURN` in Lisp, which are concerned solely with achieving data and control flow.

- The primitive operations and tests, such as `CAR`, `CDR`, `PLUS`, `NULL`, `MINUSP`, and so on, in Lisp, which perform actual computations.

Each primitive operation or test is translated into the corresponding input/output or test specification. The connective tissue primitives are then

translated into the pattern of control flow arcs, data flow arcs, and join specifications between the boxes of the plan.

In summary, the answer to the question, Is the Plan Calculus a programming language?, is *Yes* the Plan Calculus is a language with the expressive power of a wide-spectrum, very-high-level programming language, but *No* it is not necessarily appropriate for programmers to use directly.

The Evolution of Languages

A second relationship of this work to programming languages is the role of clichés in the evolution of languages. Typically, part of the advance from a lower to higher level language involves moving an entire class of decision-making from the realm of the programmer to the realm of the compiler. For example, in moving from machine language to high-level languages, the task of register allocation was moved to the compiler. As part of moving from high-level to very-high-level languages, an attempt is being made to make efficient data structure selection the responsibility of the compiler.

Another part of language evolution, however, involves identifying clichés (common patterns of usage) in the lower language, and absorbing them into the syntax of the next higher language. For example, the common patterns of jumps and tests used to perform iteration in machine language became the various looping forms of high-level languages. As part of moving from high-level to very-high-level languages, an attempt is being made to extend the syntax of languages to support common clusters of operations.

From this point of view, what it means to be a cliché is not absolute, but rather what a concept is called between the time it is identified as a common usage in the current language and the time it gets absorbed into the next higher level of language. However, this evolutionary process does not stop at the next level—as long as a language is used, new clichés will arise.

4.2 Other Formalisms

Past efforts to codify programming knowledge have used one of the following formalisms:

- program schemas [19]
- program transformations [4, 10, 12, 55]

- program refinement rules [5]
- formal grammars [47]

Although each of these representations has been found useful in certain applications, none combines all of the important features of the Plan Calculus.

Program schemas (incomplete program texts with constraints on the unfilled parts) have been used by Wirth [65] to catalog programs based on recurrence relations, by Basu and Misra [8] to represent typical loops for which the loop invariant is already known, and by Gerhart [19] and Misra [37] to represent and prove the properties of various other common forms. Unfortunately, as illustrated by the linear search example in Section 3.1, the syntax of conventional programming languages is not well suited for the kind of generalization needed in this endeavor.

Programming languages descended from Simula [13], such as CLU [30] and Alphard [52], provide a syntax for specifying standard forms, such as linear search, in a more canonical way. However, there are two more fundamental difficulties with using program schemas to represent standard program forms, which Simula and its descendants do not solve. First, programs (and therefore program schemas) are not in general easy to combine, nor are they additive. This means that when you combine two program schemas, the resulting schema is not guaranteed to satisfy the constraints of both of the original schemas, due to such factors as destructive interactions between variable assignments. Second, existing programming languages do not allow multiple views of the same program or overlapping module hierarchies. The reason for this is that, from the standpoint of these languages, a program is still basically thought of as a set of instructions to be executed, rather than as a set of descriptions (e.g., blueprints) which together specify a computation.

The most common approach for representing implementation relationships between clichés is to use knowledge-based[13] program transformation and refinement rules [5]. The major deficiency of these formalisms, as compared to overlays in the Plan Calculus, is their asymmetry between analysis and synthesis. An overlay is made up of two plans, either of which can be used as the "pattern." In a typical program synthesis step the right side plan is used as the pattern and the left side plan is instantiated as a further implementation. Conversely, in a typical analysis step, the left side plan serves

[13]As opposed to the folding-unfolding and similar transformations of Burstall and Darlington [10], which are intended to be a small set of very general transformations that must be composed appropriately to construct intuitively meaningful implementation steps.

as the pattern and the right side plan is instantiated as a more abstract description. With program transformation and refinement rules, this sort of symmetric use is not possible, since the right side is often a sequence of substitution or modification actions to be executed, rather than a declarative description that can be used as a pattern.

Another formalism used for codifying programming knowledge is formal (string) grammars. For example, Ruth [47] constructed a grammar (with global switches to control conditional expansions) which represented the class of programs expected to be handed in as exercises in an introductory PL/1 programming class. This grammar was used in a combination of top-down, bottom-up and heuristic parsing techniques in order to recognize correct and near-correct programs. Miller and Goldstein [34] also used a grammar formalism (implemented as an augmented transition network) to represent classes of programs in a domain of graphical programming with stick figures. The major shortcoming of these grammars is that they are string-based and therefore too close to the programming language.

4.3 Limitations of the Plan Calculus

This section outlines a number of known limitations of the Plan Calculus, and suggests some directions for their remedy. The Plan Calculus is just a first step in developing knowledge representations for the programming domain.

Other Kinds of Knowledge

There are at least two fundamental kinds of knowledge used in the programming task that the current Plan Calculus has no facilities to express.

One such kind of knowledge concerns the performance properties of algorithms and data structures. This kind of knowledge is used, for example, to choose between alternative implementations of a data abstraction or input/output specification. The most straightforward idea for adding this kind of information to the Plan Calculus would be to simply annotate plans with explicit performance statements, such as "this is a quadratic algorithm", and so on. However, this approach only scratches the surface of the issue. In order to make effective engineering trade-offs, a formal language is also needed for characterizing the distribution of input data to a program. Going even deeper, a representational framework is needed within which programs can be analyzed to identify bottlenecks, and within which potential optimizations

can be evaluated and compared. Recent work in this area by Kant [25, 26] starts with a program representation similar to the Plan Calculus.

A second kind of knowledge that figures prominently in many programming tasks concerns the structures and constraints of the application domain. For example, Barstow [6, 7] has studied in detail the role of mathematical models of physical processes in the synthesis of oil well log interpretation software. Since programs can be written in any domain, the problem of representing domain knowledge in programming is in principle no less general than the general problem of knowledge representation. The challenge from the point of view of the programming task, however, is how domain knowledge interacts with "computer science knowledge" (algorithms, data structures, performance properties, and so on). Neighbors [38], for example, has developed a transformation-based architecture in which domain descriptions can be formalized and combined with software implementation knowledge.

Non-Local Flow

The Plan Calculus also has limitations in expressive power within the kind of knowledge that it does address. Consider a program in which data flow is achieved by one component updating a global data base and another component querying it. Using the Plan Calculus straightforwardly, the entire data base would have to be both an input and output to every module that updated it, and an input to every module that queried it. This representation does not allow for the fact that certain modules may only produce and consume certain kinds of data, and that the intended data flow graph may therefore be significantly smaller than the straightforward data flow graph. What is suggested to solve this problem is a use of overlays in which a mutable object (such as a data base) is conceptually partitioned into several separate objects, each with a separate data flow.

A similar problem arises with the straightforward use of control flow in the Plan Calculus to model `THROW` in Lisp, or interrupt facilities in other languages. In this case, the straightforward control flow graph requires a corresponding control flow exit from every module enclosing the point of the `THROW` (or interrupt signal). Technically, this makes every enclosing module into a test specification. Conceptually, however, this seems wrong. What is needed is some way (again perhaps using overlays) of viewing the "usual" control flow separate from the interrupt-based control.[14]

[14]Harel's statecharts [23] provide a nice solution to this problem within a graphical

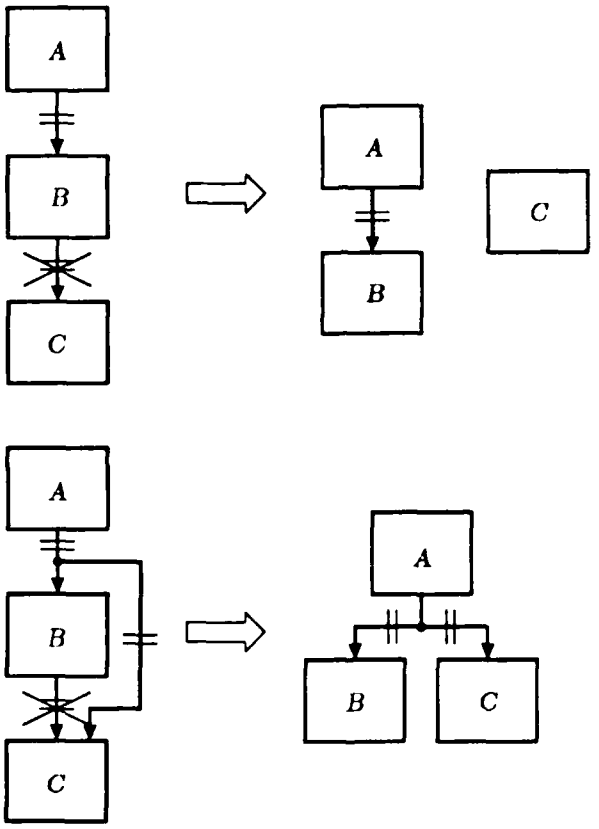


Figure 40. The two plan diagrams at the left have the same meaning. However, deletion of the same control flow arc in both plans results in two new plans (shown on the right) with different meanings.

The data and control flow problems described above may be summarized by observing that the current Plan Calculus is oriented toward representing the *local* flow of data and control. Both of the examples above are a kind of non-local flow.

Canonicalness

A desired property of the Plan Calculus is that there be a unique representation for each cliché. Since being a cliché is essentially an empirical, pre-theoretic notion, deciding whether this property holds is not a formally definable question. There is, however, a closely related formal property of the Plan Calculus which is also desired (and unfortunately, does not hold in certain cases): Syntactically distinct plans should also be semantically distinct. The reason for desiring this property is illustrated in Figure 40.

As pointed out in Section 3.3, the two plans on the left of Figure 40 have the same meaning, due to the transitivity of control flow. This is more than just a problem of elegance—the same syntactic manipulation applied to each plan can now result in two new plans with *different* meanings, as illustrated in the figure. Deleting the control flow arc between between B and C in the top plan results in a plan in which C is unordered with respect to A and B. Deleting the same arc in the bottom plan results in a plan in which C must still follow A. A similar problem arises with control flow arcs that are redundant with data flow arcs.

One solution to this problem is to canonicalize plan diagrams on the transitive closure of the control flow. Under this solution, only the bottom plan on the left of Figure 40 would be syntactically legal. A consequence of this restriction is that it would be illegal to add a control flow arc between B and C to the plan in the top-right of Figure 40—one would have to first add the arc from A to C and then from B to C. Another way to guarantee this restriction would be to automatically update the transitive closure whenever a new control flow arc is added.

An alternative approach to solving this control flow problem is to move control flow out of the structural sublanguage of the Plan Calculus, and into the logical sublanguage. This is the approach taken in the most recent implementation of the Plan Calculus [13]. This approach takes advantage of facilities in the logical reasoning engine for efficiently maintaining transitive relations, which are also needed for other purposes.

---

formalism.

4.4 Further Work

This section describes further work by the author and others that extends and builds upon the notions of inspection methods, clichés and plans. Some of this work has already been completed and is therefore only summarized here, with references to the full descriptions elsewhere. Current work in progress and future directions are also described.

Libraries of Clichés

The most important next step in this work is to use the Plan Calculus to begin in earnest the task of codifying programming clichés. The author has compiled an initial library of several hundred clichés in the area of basic techniques for manipulating symbolic data structures (see [39, 41]). This library includes:

- data abstractions, such as set, graph, mapping, list, sequence, and tree.
- operation clichés, such as addition, deletion and associative retrieval in a set, inverting a mapping, and modifying arcs in a graph.
- data structure implementation clichés, such as indexed sequence and hash table.
- clichéd algorithm fragments, such as searching, generating and accumulating.

In addition to the various kinds of overlays between these clichés, the library is organized taxonomically using two kinds of inheritance-like relationships: *specialization* and *extension*. Recalling that a plan is essentially a set of parts with constraints between them, specialization corresponds to adding constraints; extension corresponds to adding parts.

The contents of this initial library was determined primarily by the requirement of giving a complete account of the design of the hash table example of Section 2. Barstow and Green [5, 21] have codified a similar body of clichés in this same general area using a transformational formalism. One direction to continue this codification is to deepen the coverage of the library within the area of basic techniques. For example, it might be productive to work systematically through basic texts such as [27] or [2].

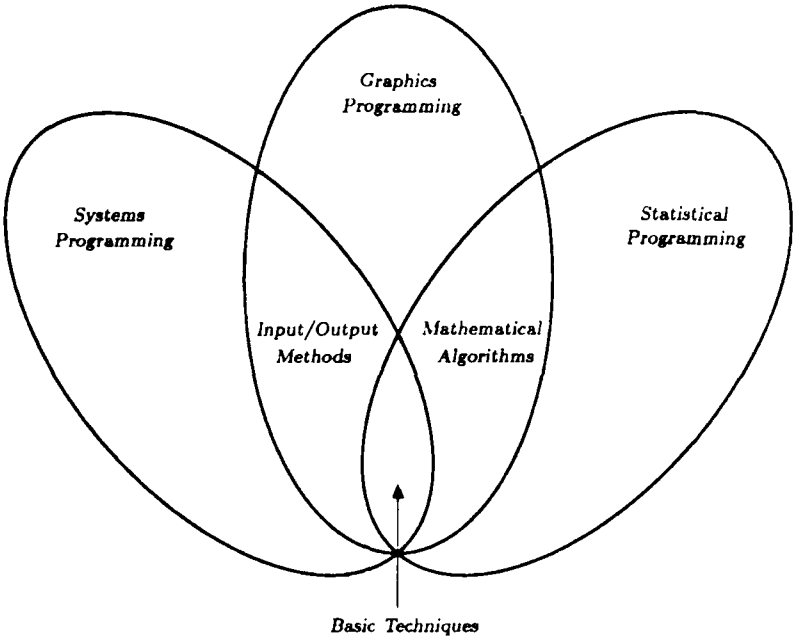


Figure 41. A Venn diagram suggesting the overlap between programming clichés used in different application areas.

A second direction to continue the codification of clichés is to broaden the coverage of the initial library toward more specialized application areas. Figure 41 suggests how this broadening might proceed from more general to more specific clichés. The figure illustrates the relationship one would expect to find between the clichés used in three areas of programming: statistics, graphics, and systems. The intersection of all three areas in the center represents basic programming techniques, where the initial codification effort has focused. The overlap between each pair of areas represents clichés of intermediate generality. The remaining part of each area represents the most specialized clichés in that area.

The Logical Sublanguage

The logical sublanguage of the Plan Calculus comprises the preconditions, postconditions and other logical statements which annotate plan diagrams. This logical language has been implemented in a reasoning system called CAKE [15, 43, 44]. CAKE supports a typed propositional logic with limited

quantificational facilities. The system includes a type inheritance lattic and special procedures for reasoning with sets, equality and other operato with common algebraic properties, such as transitivity, symmetry, and so o Side effects are modelled in the language using a situational approach simil to [32].

CAKE is a hybrid system in which manipulation of plan diagrams and rea soning in the logical sublanguage are intermixed as needed. This is achieve through an approach in which the formal semantics of data flow, control flo and other syntactic structures of plan diagrams (see [40, 41, 42]) exist a explicit logical assertions in the reasoning system's database. For exampl the semantics of a data flow arc is an equality between terms representing th appropriate ports, plus a partial order assertion between the correspondin situations.

Teleological Structure

The logical sublanguage makes it possible to talk about an important kin of structure in a plan, in addition to its control and data flow structure. Th *teleological*[15] *structure* of a plan is the set of logical relationships between th preconditions and postconditions of its input/output and test specificatio roles.

Figure 42 illustrates the concept of teleological structure with an abstra example. The figure shows an implementation overlay between a plan wit three roles, P, Q, R, and an input/output specification, S. $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{B}$, $\mathcal{E}$ etc., are formulae in the logical sublanguage, which form the precondition and postconditions of the various specifications, as shown. Data and contr flow arcs between P, Q, and R, are omitted.

In order for the overlay in Figure 42 to be valid, each postcondition of must be implied by some postcondition of P, Q, or R; and each preconditi of P, Q, and R must be implied by either a postcondition of a preceding ste or a precondition of S.[16] The pattern of these logical relationships provides deeper characterization of the purpose of each step in a plan, than is provid by control and data flow structure alone.

For example, we can see in Figure 42 that P is essentially a preparato step – all of its postconditions are prerequisites for later steps. Q and R,

[15] From the Greek *teleos*, meaning purpose. This term is first introduced in [45].

[16] The possibility that a postcondition achieved by one step may be "undone" by subsequent step is taken care of inside the logic through the use of situations.

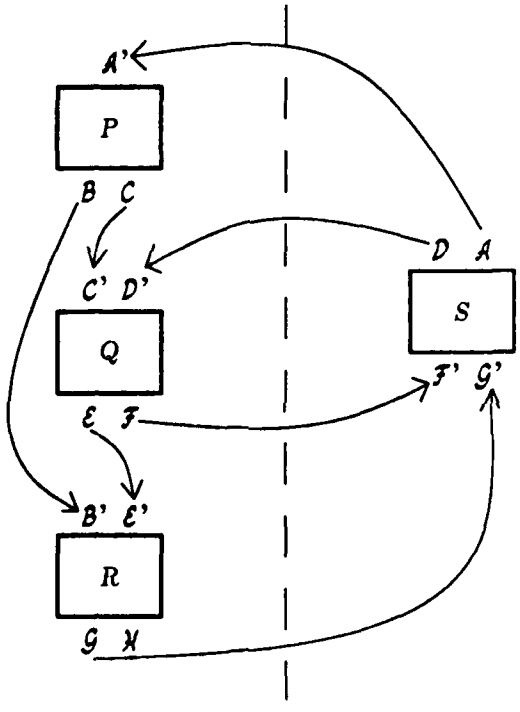


Figure 42. An overlay diagram illustrating teleological structure, i.e., implications between the respective pre- and postconditions.

the other hand, are main steps—each contributes to accomplishing part of the overall postconditions of S. (This vocabulary for describing steps of a plan in terms of their purpose is due to Goldstein [20].)

Further understanding the role of teleological structure in program analysis and synthesis is an important area for future work. For example, an analysis of the teleological structure in Figure 42 suggests that step R may be replaced by a weaker specification, since postcondition H is not needed to accomplish any part of S. In CAKE, teleological structure is represented by the dependencies in a truth-maintenance system.

The Programmer's Apprentice

The work described in this paper has evolved within the context of a project aimed at developing an intelligent, interactive assistant for software development, called the Programmer's Apprentice. The Plan Calculus serves as the "mental language" of the Apprentice.

Plan diagrams were originally developed for use in the Apprentice by Rich and Shrobe [45] and later extended by Waters [58]. Overlays, the logical sublanguage, and the formal semantics of the Plan Calculus were added by Rich [41]. The current implementation of the Plan Calculus in CAKE is only the most recent in a series of versions that have been experimented with over a period of years. As part of these experiments, modules have been implemented to translate between the Plan Calculus and an assortment of programming languages, including (subsets of) Lisp, Ada, PL/1, Fortran, and Cobol.

As part of the Programmer's Apprentice project, prototype systems has been implemented using the Plan Calculus to demonstrate both analysis and synthesis by inspection.

Wills [63] has implemented a prototype analysis by inspection system that first translates an input program into the Plan Calculus and then applies a graph parsing algorithm developed by Brotsky [9]. The grammar used in the parsing is derived from Rich's library of clichés for basic symbolic programming techniques [41, 39]. As a way of communicating the results of its analysis, Wills' system produces a kind of program explanation. Figure 43 shows the result of applying Wills' program to the `TABLE-LOOKUP` function of Section 2. Note that the convention in this explanation is that terms with initial capitals are the names of clichés or roles; terms in all capitals are identifiers in the Lisp program.

```
(DEFUN TABLE-LOOKUP (TABLE KEY)
  (LET ((BUCKET (AREF TABLE (HASH KEY TABLE))))
    (LOOP
      (IF (NULL BUCKET) (RETURN NIL))
      (LET ((ENTRY (CAR BUCKET)))
        (IF (EQUAL (KEY ENTRY) KEY) (RETURN ENTRY)))
      (SETQ BUCKET (CDR BUCKET)))))
```

```
TABLE-LOOKUP is an Associative Retrieval operation.
  If there is an element of the Set TABLE whose Key
  is KEY, then it returns it; otherwise it returns nil.
  The Key is extracted from an entry by the function KEY.
The Set is implemented as a Hash Table.
The Hash Table is implemented as an Array of Buckets,
indexed by hash code.
  The Hash Function is HASH.
  The Buckets are implemented as Lists. There are no
  header cells.  A Linear Search is used to determine
  whether or not there is an element with the given Key
  in the fetched Bucket, BUCKET.
```

Figure 43. Wills' system analyzed the undocumented Common Lisp code above and automatically produced an explanation of its implementation in terms of a library of clichés.

```
Define a linear-search program BUCKET-DELETE with
  parameters BUCKET and KEY.
Fill the enumerator with a trailing-pointer-list-enumeration
 of BUCKET.
Fill the search-criterion with (EQUAL (KEY (CAR LIST)) KEY).
Fill the action with a splice-out of PREVIOUS.
```

```
(DEFUN BUCKET-DELETE (BUCKET KEY)
  (LET* ((PREVIOUS BUCKET)
         (LIST (CDR PREVIOUS)))
    (LOOP
      (IF (NULL LIST) (RETURN NIL))
      (WHEN (EQUAL (KEY (CAR LIST)) KEY)
        (RPLACD PREVIOUS (CDDR PREVIOUS))
        (RETURN NIL))
      (SETQ PREVIOUS LIST)
      (SETQ LIST (CDR LIST)))))
```

Figure 44. Waters' system synthesized the Lisp code above from the description of the clichés to be used.

Waters [61, 60] has implemented a prototype synthesis by inspection system, called KBEMACS (for Knowledge-Based Editor in EMACS). KBEMACS allows a programmer to construct and modify programs more quickly and reliably than using a conventional program editor, by supporting operations on a program in terms of clichés. For example, Figure 44 shows a set of commands given to KBEMACS that produces a version of the BUCKET-DELETE program. The only difference between the version of BUCKET-DELETE produced by KBEMACS and the version in Section 2 is the use of an unnecessary temporary variable, LIST. This is due to the fact that the algorithm KBEMACS uses for achieving data flow using variables is not optimal.

Other Related Work

Program representations related to and derived from the Plan Calculus have been used by others in the areas of program recognition [16], programming tutors [28], program translation [14, 62], algorithm design [26], debugging [51, 31], and maintenance [33]

Soloway and Ehrlich [56] have conducted a number of empirical studies with programmers which support the psychological reality of plans and clichés.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* McGraw Hill, 1985.

[2] A. V. Aho, J. D. Ullman, and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[3] G. Arango and P. Freeman. Modeling knowledge for software development. In *Proc. 3rd Int. Wrkshp on Software Specs. and Design*, pages 63–66, London, England, 1985.

[4] R. M. Balzer. Transformational implementation: An example. *IEEE Trans. Software Engineering*, 7(1):3–13, January 1981.

[5] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12(1 & 2):73–119, 1979. PhD Thesis.

[6] D. R. Barstow. A perspective on automatic programming. *AI Magazine*, 5(1):5–27, Spring 1984.

[7] D. R. Barstow. Domain-specific automatic programming. *IEEE Trans. Software Engineering*, 11(11):1321–1336, November 1985.

[8] S. K. Basu and J. Misra. Some classes of naturally provable programs. In *2nd Int. Conf. on Software Engineering*, San Francisco, CA, 1976.

[9] D. Brotsky. *An Algorithm for Parsing Flow Graphs.* Technical Report 704, MIT Artificial Intelligence Lab., March 1984. M. S. Thesis.

[10] R. M. Burstall and J. L. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1), January 1977.

[11] D. Chapman. *Cognitive Cliches.* Working Paper 286, MIT Artificial Intelligence Lab., April 1986.

[12] T. E. Cheatham. Reusability through program transformation. *IEEE Trans. Software Engineering*, 19(5):589–595, September 1984.

[13] O. J. Dahl and K. Nygaard. SIMULA - An ALGOL-based simulation language. *Comm. of the ACM*, 9(9):671–678, September 1966.

[14] G. Faust. *Semiautomatic Translation Of COBOL into HIBOL.* Technical Report 256, MIT Lab. of Computer Science, March 1981. M.S. Thesis.

[15] Y. A. Feldman and Rich C. *The Interaction Between Truth Maintenance, Equality, and Pattern-Directed Invocation: Issues of Completeness and Efficiency.* Working Paper 296, MIT Artificial Intelligence Lab., May 1987.

[16] S. F. Fickas and R. Brooks. Recognition in a program understanding system. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pages 266-268, Tokyo, Japan, August 1979.

[17] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[18] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32, American Math. Society, Providence, RI, 1967. Vol. 19.

[19] S. L. Gerhart. Knowledge about programs: A model and case study. In *Proc. of the Int'l. Conference on Reliable Software*, pages 88–95, June 1975.

[20] I. P. Goldstein. Summary of MYCROFT: A system for understanding simple picture programs. *Artificial Intelligence*, 6(3), 1975.

[21] C. Green and D. R. Barstow. On program synthesis knowledge. *Artificial Intelligence*, 10(3):241–279, November 1978.

[22] M. T. Harandi and F. H. Young. Template based specification and design. In *Proc. 3rd Int. Wrkshp on Software Specs. and Design*, pages 94-97, London, England, 1985.

[23] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[24] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–583, October 1969.

[25] E. Kant. On the efficient synthesis of efficient programs. *Artificial Intelligence*, 20(3):253-306, May 1983.

[26] E. Kant. Understanding and automating algorithm design. *IEEE Trans. Software Engineering*, 11(11):1361-1374, November 1985.

[27] D. E. Knuth. *The Art of Computer Programming.* Volume 1, 2, 3, Addison Wesley, 1968, 1969, 1973.

[28] J. Laubsch and M. Eisenstadt. Domain specific debugging aids for novice programmers. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 964-969, Vancouver, British Columbia, Canada, August 1981.

[29] J. Z. Lavi. Improving the embedded computer systems software process using a generic model. In *Proc. 3rd Int. Wrkshp on Software Specs. and Design*, pages 127-129, London, England, 1985.

[30] B.H. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* McGraw Hill, 1986.

[31] R. Lutz. *Program Debugging by Near-Miss Recognition and Symbolic Evaluation.* Technical Report CSRP.044, U. of Sussex, 1984.

[32] Z. Manna and R. Waldinger. The deductive synthesis of imperative LISP programs. In *Proc. 6th National Conf. on Artificial Intelligence*, pages 155-160, Seattle, WN, July 1987.

[33] B. P. McCune and J. S. Dean. *Advanced Tools for Software Maintenance.* Technical Report 313, Rome Air Development Ctr., Griffiss AFB, NY 13441, December 1982.

[34] M. L. Miller and I Goldstein. Problem solving grammars as formal tools for intelligent CAI. In *Proc. of the Association for Computing Machinery*, 1977.

[35] M. Minsky. *Society of Mind.* Simon & Schuster, January 1987.

[36] M. L. Minsky. A framework for representing knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision*, McGraw Hill, 1975.

[37] J. Misra. A technique of algorithm construction on sequences. *IEEE Trans. Software Engineering*, 4(1):65–69, January 1978.

[38] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Trans. Software Engineering*, 10(5):564–574, September 1984.

[39] C. Rich. Inspection methods in programming: A library of basic plans. In preparation.

[40] C. Rich. The Plan Calculus: Logical foundations. In preparation.

[41] C. Rich. *Inspection Methods in Programming.* Technical Report 604. MIT Artificial Intelligence Lab., June 1981. PhD Thesis.

[42] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044–1052, Vancouver, British Columbia, Canada, August 1981. Reprinted in M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, Springer Verlag, 1984 and in C. Rich and R. C. Waters, editors,*Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[43] C. Rich. Knowledge representation languages and predicate calculus: How to have your cake and eat it too. In *Proc. 2nd National Conf. on Artificial Intelligence*, Pittsburgh, PA, August 1982.

[44] C. Rich. The layered architecture of a system for reasoning about programs. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 540–546, Los Angeles, CA, 1985.

[45] C. Rich and H. E. Shrobe. Initial report on a LISP Programmer's Apprentice. *IEEE Trans. Software Engineering*, 4(6), November 1978. Reprinted in D. Barstow, E. Sandewall, and H. Shrobe, editors, *Interactive Programming Environments*, McGraw-Hill, 1984.

[46] C. Rich and R. C. Waters. *Abstraction, Inspection and Debugging in Programming.* Memo 634, MIT Artificial Intelligence Lab., June 1981.

[47] G. R. Ruth. Intelligent program analysis. *Artificial Intelligence*, 7:65–85, 1976.

[48] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[49] E. D. Sacerdoti. The nonlinear nature of plans. In *Proc. 4th Int. Joint Conf. Artificial Intelligence*, pages 206–214, Tblisi, Georgia, USSR, September 1975.

[50] R. C. Schank and R. P. Abelson. *Scripts, Plans, Goals, and Understanding.* Lawrence Erlbaum, 1977.

[51] D. Shapiro. *SNIFFER: A System that Understands Bugs.* Memo 638, MIT Artificial Intelligence Lab., June 1981. M.S. Thesis.

[52] M. Shaw, W. A. Wulf, and R. L. London. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Comm. of the ACM*, 20(8):553–563, August 1977.

[53] H. E. Shrobe. *Dependency Directed Reasoning for Complex Program Understanding.* Technical Report 503, MIT Artificial Intelligence Lab., April 1979. PhD Thesis.

[54] H. E. Shrobe. Common sense reasoning about side effects to complex data structures. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, Tokyo, Japan, August 1979.

[55] D. R. Smith, G. B. Kotik, and S. J. Westfold. Research on knowledge-based software engineering environments at Kestrel Institute. *IEEE Trans. Software Engineering*, 11(11):1278–1295, November 1985.

[56] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Engineering*, 10(5):595–609, September 1984.

[57] G. J. Sussman. Slices at the boundary between analysis and synthesis. In J. C. Latombe, editor, *Artificial Intelligence and Pattern Recognition in Computer-Aided Design*, North Holland, March 1978. Proc. of the IFIP Working Conference, Grenoble, France.

[58] R. C. Waters. *Automatic Analysis of the Logical Structure of Programs.* Technical Report 492, MIT Artificial Intelligence Lab., December 1978. PhD Thesis.

[59] R. C. Waters. A method for analyzing loop programs. *IEEE Trans. Software Engineering*, 5(3):237–247, May 1979.

[60] R. C. Waters. *KBEmacs: A Step Towards the Programmer's Apprentice* Technical Report 753, MIT Artificial Intelligence Lab., May 1985.

[61] R. C. Waters. The Programmer's Apprentice: A session with KBEmacs. *IEEE Trans. Software Engineering*, 11(11):1296–1320, November 1985.

[62] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. Software Engineering*, Summer 1987. (To appear).

[63] L. M. Wills. *Automated Program Recognition.* Technical Report 904, MIT Artificial Intelligence Lab., September 1986. M.S. Thesis.

[64] P. H. Winston. Learning structural descriptions from examples. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 157–210. McGraw-Hill, 1975.

[65] N. Wirth. *Systematic Programming, An Introduction.* Prentice Hall, 1973.

END

DATE

FILMED

6-1988

DTIC